Diplomarbeit

# Graph Modification Problems and Automated Search Tree Generation

Falk Hüffner

|  |  |
|---|---|
| **Betreuer:** | PD Dr. Rolf Niedermeier |
|  | Dr. Jens Gramm |
|  | Dipl.-Inform. Jiong Guo |

**begonnen am:** 24. April 2003

**beendet am:** 7. Oktober 2003

NWG Theoretische Informatik/Parametrisierte Algorithmen
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

**Abstract.** We present a framework for the automated generation of exact search tree algorithms for *NP*-hard problems. The purpose of our approach is two-fold: rapid development and improved upper bounds.

Many search tree algorithms for various problems in the literature are based on complicated case distinctions. Our approach may lead to a much simpler process of developing and analyzing these algorithms. Moreover, using the sheer computing power of machines it can also provide improved upper bounds on search tree sizes (i.e., faster exact solving algorithms) in comparison with previously developed "hand-made" search trees. The generated search tree algorithms work by expanding a "window view" of the problem with local information, and then branch according to a precalculated rule for the resulting window.

A central example in this work is given with the *NP*-complete CLUSTER EDITING problem, which asks for the minimum number of edge additions and deletions in a graph to create a cluster graph (i.e., a graph which is a disjoint union of cliques). For CLUSTER EDITING, a hand-made search tree is known with $O(2.27^k)$ worst-case size, which is improved to $O(1.92^k)$ due to our new method. (Herein, $k$ denotes the number of edge modifications allowed.) A refinement of the search tree generator is presented, which tries to direct the expansion of the window towards advantageous case sets. The scheme is also successfully applied to several other graph modifications problems; generalizations to other problems are sketched.

# Contents

Chapter 1

# Introduction

Many important real-world problems are *NP*-hard, which means there are no known polynomial-time algorithms to solve them. Several approaches have been proposed to attack these problems, among them approximation algorithms and heuristic algorithms. Often, however, it is important to find *exact* (i.e., optimal) solutions. While all known more refined exact algorithms for *NP*-hard problems still run in super-polynomial time, they can show vast differences in both theoretical and practical performance.

A recent survey by Woeginger [Woe03] examines various approaches to exact algorithms for *NP*-hard problems. One of the most common is *pruning the search tree*, outlined as follows.

Usually, a feasible solution of an optimization problem can be split into several "pieces"; for example, for *subset problems*, every feasible solution can be specified as a subset of an underlying ground set, and we can for each element of the ground set consider separately whether it is part of the solution.

Every *NP*-complete optimization problem can be solved by exhaustively enumerating all feasible solutions and checking for the optimal one. The enumeration can be organized in a *search tree*:

- Consider some piece of the feasible solution;
- Determine all possible values of this piece (for example for subset problems, two values: element of the set or not);
- Branch recursively into several subcases according to these possible values.

The gain over plain enumeration in this approach comes from the possibility to recognize values for certain pieces of the solution that cannot be part of any optimal solution. In those cases, the whole subtree below the corresponding branch can be pruned, leading to a potentially large speedup. The tools that will be introduced in Sect. 1.4 provide a means for worst-case runtime analysis of these search trees.

Using this approach, many exact algorithms for well-known *NP*-complete problems have been devised. For instance, search tree based algorithms have been developed for

- Satisfiability [Kul99, Hir00],
- Maximum Satisfiability [BR99, NR00b, CK02, GHNR03],
- Exact Satisfiability [DP02, HK02],
- Independent Set [DJ02, Rob86, Rob01],
- Vertex Cover [CKJ01, NR03b],
- Constraint Bipartite Vertex Cover [FN01]
- 3-Hitting Set [NR03a],

and numerous other problems.

Most of these algorithms have undergone some kind of "evolution" towards better and better worst-case bounds on their running times. These improved bounds, however, usually come at the cost of distinguishing between more and more combinatorial cases, which makes the development and the correctness proofs a tedious and error-prone task. For example, in a series of papers the upper bound on the search tree size for an algorithm solving Maximum Satisfiability was improved from $1.62^K$ [MR99] to $1.38^K$ [NR00b] to $1.34^K$ [BR99], and recently to $1.32^K$ [CK02], where $K$ denotes the number of clauses in the given formula in conjunctive normal form.

As Hirsch and Kulikov [HK02] recently observed in the context of satisfiability problems, it would be interesting to design a computer program that outputs *mechanically* proven worst-case upper bounds based on simple combinatorial reduction rules that lead to nontrivial and useful search tree algorithms. In this work, we present such an automated approach for the development of efficient search tree algorithms, focusing on *NP*-hard graph modification problems.

Our work may be considered as a special case of algorithm engineering. We present programs not to solve decision or optimization problems, but to develop efficient programs (i.e., search tree algorithms with "small" exponential worst-case running time).

Our approach is based on the separation of two tasks in the development of search tree algorithms—namely, on the one hand, the investigation and development of clever problem-specific rules (this is usually the creative, thus, the "human part") and, on the other hand, the analysis of numerous cases using these problem-specific rules (this is the "machine part"). The software environment we deliver can also be used in an interactive way in the sense that it points the user to the worst case in the current case analysis. Then, the user may think of additional problem-specific rules to improve this situation, obtain a better bound, and repeat the process.

The automated generation of search tree algorithms in this work deals mainly with the class of graph modification problems [Cai96, LY80, NSS01], although the basic ideas appear to be generalizable to other graph and even non-graph problems. In particular, we study the following *NP*-complete edge modification problem CLUSTER EDITING, which is motivated by, e.g., data clustering applications in computational biology [Sha02, SST02] and correlation analysis in machine learning [BBC02]:

> **Input:** An undirected graph $G = (V, E)$ and a nonnegative integer $k$.
>
> **Question:** Can we transform $G$, by deleting and adding at most $k$ edges, into a graph that consists of a disjoint union of cliques?

In Sect. 2.5, we will present a search tree based algorithm exactly solving CLUSTER EDITING in $O(2.27^k + |V|^3)$ time. This algorithm is based on case distinctions developed by "human case analysis" and it took us about three months of development and verification. Now, based on some relatively simple problem-specific rules (whose correctness is easy to check), we obtain an $O(2.16^k + |V|^3)$ time algorithm for the same problem. This is achieved by an automated case analysis that checks much more subcases than we were able to do "by hand". With additional problem-specific rules, we can even achieve $O(1.92^k + |V|^3)$ time.

Altogether (including computation time on a single Linux PC and the development of the reduction rules), using our mechanized framework this significantly improved running time for an exact solution of CLUSTER EDITING could be achieved in about one week.

The example application to CLUSTER EDITING exhibits the power of our approach, whose two main potential benefits we see as

1. rapid development and
2. improved upper bounds

due to automation of tiresome and more or less schematic but extensive case-by-case analysis. Thus, we hope that this work contributes a new way to relieve humans from tedious and error-prone work. Besides CLUSTER EDITING, we present applications of our approach to other *NP*-complete graph modification (i.e., edge or vertex deletion) problems including CLUSTER DELETION (the special case of CLUSTER EDITING where only edge deletions are allowed) and the generation of triangle-free graphs and cographs. Additionally, we present initial results for BOUNDED DEGREE 3 DOMINATING SET.

Some parts of the results from Chap. 3–5 were presented in [GGHN03a].

## 1.1 Structure of the Work

In the remaining parts of this chapter, in Sect. 1.2, we will first introduce some basic notations and definitions, mainly from graph theory. Then, in Sect. 1.3.2, we discuss the computation of exact solutions for *NP*-hard problems. Basic tools for mathematical analysis of the performance of our algorithms are recurrences and *branching vectors*; they are introduced in Sect. 1.4. Section 1.5 gives a brief introduction to the Ocaml programming language, which was used to implement our framework.

Chapter 2 introduces and motivates the problem class of *graph modification problems*, which ask for the minimal amount of changes to the edge or vertex set of an input graph needed to obtain a graph with a certain property. The focus is on Cluster Editing (Sect. 2.2), which is motivated from computational biology, machine learning and other areas. In Sect. 2.2.2, we devise a simple parameterized search tree algorithm for Cluster Editing. The concept of *annotations*, which can capture additional information over the course of a search tree algorithm, is introduced in Sect. 2.3. In Sect. 2.4, we make a brief excursion to a *problem kernel reduction* for Cluster Editing, a technique valuable to reduce the size of an instance of a parameterized problem before applying further solving techniques. In Sect. 2.5 then, we present the algorithm which gave the main incentive for the development of the automation framework: A refined search tree algorithm for Cluster Editing. The treatment of Cluster Editing is rounded off by devising some heuristic improvements in Sect. 2.6.

Chapter 3 eventually introduces the idea of automating the quest for efficient algorithms. The refined Cluster Editing algorithm from Sect. 2.5, when slightly simplified, is identified as an element of a large set of algorithms (Sect. 3.1). We design a method to find elements of these sets which are optimal in Sect. 3.2. Section 3.3 describes some optimizations which are essential to find results within reasonable time. The results for Cluster Editing presented in Sect. 3.4 prove the validity of the approach with an algorithm noticeably improving upon the manually found one. The applicability of our algorithm for finding branching rules is not limited to Cluster Editing; we give a detailed description of the application to a large class of graph modification problems in Sect. 3.5.

In Chap. 4, we extend the set of considered search trees by examining algorithms that not only vary in the rules for the branching of each case considered, but also vary the set of cases itself. To get more exploitable structure in the input graphs, we can identify "trivial" instances of the problem, and assume them to be eliminated, thereby creating invariants which can be utilized to restrict the set of generated cases. We devise such a rule for Cluster Editing in Sect. 4.1, and then go on to describe how to find optimal search tree algorithms using this rule in Sect. 4.2. In Sect. 4.3, we also give a proposition which allows the application of this scheme to several further graph modification problems.

In Chap. 5, we present a variety of experimental results of applications of our frameworks to graph modification problems. We discuss edge deletion (Sect. 5.1) and vertex deletion (Sect. 5.2) problems. With BOUNDED DEGREE DOMINATING SET, we also demonstrate an application to a problem which is not a graph modification problem (Sect. 5.3). A summary (Sect. 5.4) shows improvements over known results for the discussed problems.

Several important details of the implementation are explained in Chap. 6. The efficient representation of graphs is shown in Sect. 6.1; Sect. 6.2 deals with the efficient handling of sets of branching vectors, which is a major computational bottleneck for our framework.

We conclude with a brief summary and an outlook to further work in Chap. 7.

## 1.2 Preliminaries and Basic Notation

All graphs discussed in this work are assumed to be undirected, simple, and without self-loops. We call a graph $G' = (V', E')$ *induced subgraph* of a graph $G = (V, E)$ iff $V' \subseteq V$ and $E' = \{\{u, v\} \mid u, v \in V' \text{ and } \{u, v\} \in E\}$.

A *graph property $\Pi$* is simply a mapping from the set of graphs to *true* and *false*. If a graph $G$ maps to *true* under $\Pi$, one says that "$G$ has property $\Pi$", or "$G$ is a $\Pi$-graph". A property is *nontrivial* if the set of graphs with the property and the set of graphs without the property are both infinite. If for any graph $G$ with property $\Pi$ every induced subgraph of $G$ is also a $\Pi$-graph, then $\Pi$ is a *hereditary* property. A property $\Pi$ has a *forbidden set characterization* if there is a set of graphs $F$ such that a graph is a $\Pi$-graph iff it contains no graph from $F$ as induced subgraph. Note that clearly every property with forbidden set characterization is hereditary; also any hereditary property has a forbidden set characterization, but the forbidden set is not necessarily finite [Cai96].

If $x$ is a vertex in a graph $G = (V, E)$, then by $N_G(x)$ we denote the set of its neighbors, i.e., $N_G(x) := \{v \mid \{x, v\} \in E\}$. The *degree* of a vertex is the number of neighbors: $\deg_G(x) := |N_G(x)|$. We omit the index if it is clear from the context. With *vertex pair*, we always mean an unordered pair of distinct vertices, i.e., $\{x, x\}$ is not a vertex pair. A *clique* is a complete graph, i.e., a graph $G = (V, E)$ where for all $u, v \in V$ the edge $\{u, v\}$ is in $E$.

We use the notation $G \setminus W$ for a set $W \subseteq V$ to denote the induced subgraph of $G$ where all vertices in $W$ are deleted, i.e., $G \setminus W := (V \setminus W, \{\{u, v\} \in E \mid u \notin W \wedge v \notin W\})$.

For a set $A$ and an element $x$, we write $A + x$ for $A \cup \{x\}$ and $A - x$ for $A \setminus \{x\}$. For two sets $E$ and $F$, $E \bigtriangleup F$ is the symmetric difference of $E$ and $F$, i.e., $(E \setminus F) \cup (F \setminus E)$.

We use standard notation for special graphs: $P_n$ is a path of $n$ vertices;

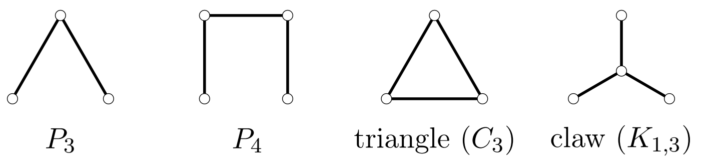

Figure 1.1: Some special graphs

$C_n$ is a cycle of $n$ vertices; and $K_{n,m}$ is the complete bipartite graph with partitions of size $n$ and $m$ (see also Fig. 1.1).

## 1.3 Exact Solutions for *NP*-Hard Problems

We will show two approaches to analyze and improve the performance of exact algorithms for *NP*-hard problems.

### 1.3.1 Improved Exponential Bounds

As Woeginger notes [Woe03], the quality of exact algorithms for *NP*-hard problems is sometimes hard to compare, since their analysis is done in terms of different parameters. He recommends the approach of including an explicit measure for the problem size in the problem specification. For instance, for the Maximum Independent Set problem, we can decide to do the analysis in terms of the number of vertices $n$, instead of in terms of the number of edges or other properties of the input.

One can then usually formulate a trivial algorithm which runs in $O(c^n)$ time. The basic goal is to improve upon the constant $c$, i.e., to devise algorithms with a smaller exponential base. Our framework to be presented offers the possibility to automate important parts of this search.

### 1.3.2 Parameterized Complexity

One of the newer areas in complexity theory is to study parameterized complexity [DF99]. It is based on the observation that for many hard problems, the apparently unavoidable combinatorial explosion can be confined to a “small part” of the input, the *parameter*, so that the problems can be solved in polynomial time when the parameter is fixed. For instance, the *NP*-complete Vertex Cover problem can be solved by an algorithm with $O(1.3^k + kn)$ running time [CKJ01, NR99, NR03b], where the parameter $k$ is a bound on the maximum size of the vertex cover set we are looking for and $n$ is the number of vertices in the given graph. As can be easily seen, this yields an efficient, practical algorithm for small values of $k$. The parameterized problems that have algorithms of time complexity $f(k)\cdot n^{O(1)}$ are called *fixed-parameter tractable* with respect to parameter $k$, where $f$ can be an arbitrary function depending only on $k$, and $n$ denotes the overall input size. An algorithm having this time complexity is called *fixed-parameter algorithm*. For details, we refer the reader to the monograph by Downey and Fellows [DF99] and the various survey papers on this field [AGN01, Fel02, Dow03].

Our framework is able to generate fixed-parameter algorithms; for example, one algorithm found for Cluster Editing runs in $O(1.92^k + |V|^3)$ time, where $k$ is the number of edge modifications, and is therefore a fixed-parameter algorithm with respect to parameter $k$.

## 1.4 Branching Vectors

Our bounded search tree algorithms work recursively. The number of recursions is the number of nodes in the corresponding search tree. This number is governed by homogeneous, linear recurrences with constant coefficients. It is well-known how to solve them and the asymptotic solution is determined by the roots of the characteristic polynomial (cf. Kullmann [Kul99] for more details).

If the algorithm solves a problem of "size" $s$ and calls itself recursively for problems of sizes $s - d_1, \dots, s - d_i$, then $(d_1, \dots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence

$$t_s = t_{s-d_1} + \cdots + t_{s-d_i},$$

where $t_s$ denotes the number of leaves in the search tree solving an instance of size $s$ and $t_j = 1$ for $0 \leq j < d$ with $d = \max(d_1, \dots, d_i)$.

This recurrence corresponds to the *characteristic polynomial*

$$z^d - z^{d-d_1} - \cdots - z^{d-d_i}.$$

The characteristic polynomial has a single root $\alpha$ which has maximum absolute value and is a positive real; then, $t_s$ is $O(\alpha^s)$. We call $\alpha$ the *branching number* that corresponds to the branching vector $(d_1, \dots, d_i)$.

Consider a parameterized problem with parameter $k$ and a search tree algorithm for it, consisting of several branching rules. For each branching rule, we can give a branching vector which has as many elements as the rule branches; each element states the amount by which the parameter is decreased for that branch. The size of the search tree is then $O(\alpha^k)$, where $k$ is the parameter and $\alpha$ is the largest branching number that will occur. For example, in Sect. 2.5, we will give an algorithm for Cluster Editing where this branching number is about 2.27; it belongs to the branching vector $(1, 2, 2, 3, 3)$.

## 1.5 A Brief Introduction to Objective Caml

Our implementation of the framework for automated generation of search tree algorithms was implemented mainly in the Objective Caml (Ocaml) language [LVD$^+$96]. Ocaml is a general purpose programming language that combines functional, imperative, and object-oriented programming. It belongs to the ML family of programming languages and has been implemented at INRIA Rocquencourt within the "Projet Cristal" group.

The language is statically typed; its type system ensures the correct evaluation of programs. Types are automatically inferred. Code can be interpreted interactively, compiled to byte code, or compiled to native code for a wide variety of platforms. Several key points proved to be very advantageous for our purposes:

- Excellent support for *functional* style programming. A purely functional program is just a single expression to be evaluated; there are no side effects or assignments, evaluating a function multiple times will always yield the same result. Programming in functional style makes code easier to write, understand, and debug. This facilitated rapid prototype development.
- Safety. An Ocaml program cannot "crash"; the type system and the run time system eliminate common errors like buffer overflows, uninitialized variables, or type cast errors. The automatic memory management (garbage collector) also removes the burden of having to manage memory manually, and eliminates the bugs related to that, like memory leaks and dangling pointers. This is very advantageous for the implementation of branching vector sets (cf. Sect. 6.2), where tree-like structures are created, joined and discarded frequently.
- User-definable data-types: The user can define new recursive datatypes as a combination of record and variant types. These types are very useful to represent recursive algorithms with several alternatives for each step.
- Finally, a high quality native code compiler producing very fast code. Since our programs run up to several days, this is very important. The standard Ocaml implementation also allows easy access to functions written in C for low-level tasks, which we used for interfacing with the *nauty* library [McK90].

Since Ocaml allows to express complicated control flow clearly and compactly, we also use it for the pseudo-code in this work; all pseudo-code is valid Ocaml. To make these examples accessible to the reader with few experience in functional languages, we will now give a short introduction to the Ocaml language. We assume familiarity with at least one imperative language, like C or Pascal.

### 1.5.1 Expressions and Functions

We will present the basic features of the Ocaml language largely by examples. For a more thorough introduction, we refer to the extensive book by Chailloux, Manoury and Pagano [CMP00]; a reference of syntax, semantics and standard library is available online [LDG$^+$02].

In addition to the traditional edit-compile-run model, Ocaml can also be run interactively: The user types phrases, terminated by `;;`, in response to the `#` prompt, and the system evaluates them and prints their type and value:

```
# 1 + 2 * 3;;
- : int = 7
```

The answer of the Ocaml system consists of three parts: The identifier to which the value is bound (none (`-`) in this example), the type of the expression (`int`, i.e., integer), and its evaluated value (here simply the result of the arithmetic operations).

The `let` keyword is used to bind values to identifiers:

```
# let a = 3;;
val a : int = 3
# a;;
- : int = 3
```

Here `val a :` shows that the value was bound to the identifier `a`.

The `fun` keyword creates unnamed functions. In the notation for a type, $t_1$ `->` $t_2$ denotes the type of a function which maps values of type $t_1$ to values of type $t_2$. The expression which defines the function is compiled to an internal format and printed as `<fun>`:

```
# let f = fun x -> x * x;;
val f : int -> int = <fun>
```

As common in mathematics, function application is done by simply giving the argument after the function:

```
# (fun x -> x * x) 3;;
- : int = 9
# let f = fun x -> x * 2;;
val f : int -> int = <fun>
# f 17;;
- : int = 34
```

Note that unlike in many other languages, no parentheses are required around the argument. As with trigonometric functions, `sin x + cos x` means `(sin x) + (cos x)`. Parentheses *are* required if the argument contains arithmetics, as in `sin (x + 2)`.

For simplicity, the type system of Ocaml only knows functions with exactly one parameter. Functions with more than one parameter can be "emulated" by so-called *currying*:

```
# let f = fun x -> (fun y -> x + y * y);;
val f : int -> int -> int = <fun>
# f 1;;
- : int -> int = <fun>
# f 1 4;;
- : int = 17
```

The type `int -> int -> int` is to be read as `int -> (int -> int)`, i.e., a function which takes an integer and returns a function taking another integer and returning an integer. Function application is right-associative, i.e., `f 1 4` is evaluated as `(f 1) 4`. Evaluating `f 1` first returns a function which maps an integer `y` to `1 + y * y`; passing `4` to this function yields the final value of `17`.

Fortunately, the `let` keyword provides "syntactic sugar" for defining functions with single or multiple arguments by currying:

```
# let f x y = x + y * y;;
val f : int -> int -> int = <fun>
```

Bindings can also be local with the `let` *id* = *expr1* `in` *expr2* construction. It allows to use *id* as a shortcut for *expr1* only in *expr2*.

```
# let f x y = let square x = x * x in square x + square y;;
val f : int -> int -> int = <fun>
# f 3 4;;
- : int = 25
```

Note that it is valid and commonly used to *shadow* bindings (`x` in this case): identifiers always refer to the innermost binding.

Recursive functions are defined with the `let rec` binding:

```
# let rec fib n = if n < 2 then 1
                           else (fib (n - 1)) + (fib (n - 2));;
val fib : int -> int = <fun>
# fib 30;;
- : int = 1346269
```

Note that the `if` expression does not denote the alternative execution of two statements, but is just an expression with a value dependent on the condition; it is equivalent to the `?:` operator in C rather than to the `if` of that language.

List handling is built-in in Ocaml. Lists are either given as a bracketed list of semicolon-separated elements, or constructed from other lists by adding elements in front using the infix `::` operator:

```
# let l = [6; 17; 23; 39];;
val l : int list = [6; 17; 23; 39]
# 3 :: l;;
- : int list = [3; 6; 17; 23; 39]
```

The standard Ocaml library provides a `List.map` function that applies a given function to each element of a list, and returns the list of the results.

```
List.map (fun n -> n * 2 + 1) [0; 1; 2; 3; 4];;
- : int list = [1; 3; 5; 7; 9]
```

Another powerful list manipulation function is *folding*. It is often used where imperative languages would utilize loops. For a function `f`, an initial value `a` and a list `[`$b_1$`; ...; `$b_n$`]`, the call `List.fold_left f a [`$b_1$`; ...;` $b_n$`]` returns `f (... (f (f a `$b_1$`) `$b_2$`) ...) `$b_n$:

```
# List.fold_left (fun s x -> s + x) 0 [1; 2; 3; 4; 5];;
- : int = 15
```

## 1.5.2 User-defined Data Structures

Tuples can simply be constructed with commas. Tuple types are denoted with asterisks (`*`):

```
# 3, "green", 2.71;;
- : int * string * float = (3, "green", 2.71)
```

Like most languages, Ocaml also has record types with named fields, analogous to C `struct`s:

```
# type ratio = {num: int; denum: int};;
type ratio = { num : int; denum : int; }
```

Values of record types are created with curly braces containing assignments of each component:

```
# {num = 1; denum = 3};;
- : ratio = {num = 1; denum = 3}
# let add_ratio r1 r2 =
   {num = r1.num * r2.denum + r2.num * r1.denum;
    denum = r1.denum * r2.denum};;
val add_ratio : ratio -> ratio -> ratio = <fun>
# add_ratio {num = 1; denum = 3} {num = 2; denum = 5};;
- : ratio = {num = 11; denum = 15}
```

Another data structure, the *variant record*, is useful to represent a set of cases of a particular type, which makes it especially suitable for recursive data structures, like binary trees. Cases are delimited with `|`. Each case has a constructor, which starts with a capital letter; optionally, an argument can be declared after the `of` keyword. Variant records are constructed simply by stating the argument after the name of the constructor:

```
# type misc = Nothing | Int of int | Tuple of int * int;;
type misc = Nothing | Int of int | Tuple of int * int
# Tuple (2, 3);;
- : misc = Tuple (2, 3)
```

In addition to these features, Ocaml provides exceptions for signaling and handling exceptional conditions, a powerful module system and support for imperative or object-oriented style programming.

# Chapter 2

# Search Tree Algorithms for Cluster Editing

In this chapter, we will examine the *NP*-hard Cluster Editing problem and develop a parameterized search tree algorithm to solve it efficiently. This will help in establishing the basic mechanisms on which we build our generalized and automated framework in the following chapters.

## 2.1 Graph Modification Problems

*Graph modification problems* [NSS01, Sha02] call for making the least number of changes to the edge and/or vertex set of an input graph in order to obtain a graph with a desired property $\Pi$. Graph Modification Problems play an important role in graph theory; the classic work on *NP*-completeness by Garey and Johnson from 1979 already mentions 18 different types of vertex and edge modification problems [GJ79, Section A1.2].

A typical motivation of graph modification problems is analysis of experimental data, where the modifications of the graphs should compensate for measuring errors; this will be explained in detail in the following section. Another application is modification of graphs to get an instance from a graph family for which the given problem is easily solvable. For instance, Leizhen Cai [Cai03] devised a parameterized algorithm that solves Vertex Coloring efficiently for graphs that are "nearly" split graphs.

Given a graph property $\Pi$ and a graph $G = (V, E)$, we can define the following problems:

**Definition 2.1.** *$\Pi$-Editing*: Find a minimum set $F \subseteq V \times V$ such that $G' = (V, E \bigtriangleup F)$ satisfies $\Pi$.

Important variations are *$\Pi$-Deletion*, where only edge deletions are allowed (i.e., $F \subseteq E$), and *$\Pi$-Completion*, where only edge additions are allowed (i.e., $F \cap E = \varnothing$).

We also consider the problem of deleting vertices:

**Definition 2.2.** *$\Pi$-Vertex Deletion*: Find a minimum set $W \subseteq V$ such that $G' := G \setminus W$ satisfies $\Pi$.

For many graph properties, these problems are *NP*-hard [NSS01]. In particular, it is a well-known result that vertex deletion problems, also known as maximum induced subgraph problems, are *NP*-hard for any nontrivial hereditary property [LY80].

## 2.2 Clustering

There is a huge variety of clustering algorithms with applications in numerous fields, e.g. computational biology [HJ97], VLSI design [HK92], and image processing [WL93]. In general, given are a set of objects and a measure of similarity between objects. The goal is to partition the objects into disjoint subsets (*clusters*), such that similarity within a cluster is high, and similarity between clusters is low. Many variations of this problem have been studied, for instance with different definitions of the optimality of a solution, or with additional constraints like a fixed number of clusters.

Here, we focus on problems closely related to algorithms for clustering gene expression data obtained with DNA microarrays (cf. [SS02] for a recent survey). More precisely, Shamir, Sharan and Tsur [SST02] recently studied three problems called CLUSTER EDITING, CLUSTER DELETION, and CLUSTER COMPLETION. These are based on the notion of a *similarity graph* whose vertices correspond to data elements and in which there is an edge between two vertices iff the similarity of their corresponding elements exceeds a predefined threshold. Finding a clustering now can be modeled as a graph modification problem where the target graphs are *cluster graphs*, which are defined as follows:

**Definition 2.3.** A *cluster graph* is a graph in which each of the connected components is a clique.

Thus, we arrive at the following graph modification problem (formulated as decision problem):

**Definition 2.4.** *Cluster Editing*: Given a graph $G = (V, E)$ and a nonnegative integer $k$. Can we transform $G$, by deleting and adding at most $k$ edges, into a graph that consists of a disjoint union of cliques (a *clustering solution*)?

An important advantage of this and related formulations for clustering problems is that one does not need to specify the number of clusters, nor a distance threshold in advance; these parameters will be implicitly chosen by the problem definition to be optimal.

### 2.2.1 Previous Results

The first proof of the *NP*-completeness of CLUSTER EDITING can be extracted from a paper by Křivánek and Morávek [KM86]. They prove the *NP*-completeness of the HIERARCHICAL-TREE CLUSTERING problem via the special case with binary weights and tree height 3 ($^b\mathrm{HIC}_3$). This special case is equivalent to CLUSTER EDITING. The *NP*-completeness of CLUSTER DELETION was proven by Natanzon [Nat99].

Shamir et al. [SST02] give, among other things, another proof that CLUSTER EDITING is *NP*-complete and then show that there exists some constant $\varepsilon > 0$ such that it is *NP*-hard to approximate CLUSTER DELETION to within a factor of $1 + \varepsilon$ (i.e., CLUSTER DELETION is *APX*-hard). CLUSTER COMPLETION is easily solvable in polynomial time by completing connected components of the given graph.

In addition, Shamir et al. study cases where the number of clusters (i.e., cliques) is fixed. Before that, Ben-Dor, Shamir and Yakhini [BDSY99] and Sharan and Shamir [SS00] investigated closely related clustering applications in the computational biology context, where they deal with somewhat modified versions of the CLUSTER EDITING problem together with heuristic polynomial-time solutions.

Independently from Shamir et al., Bansal, Blum, and Chawla [BBC02] study CLUSTER EDITING motivated by document clustering problems from machine learning. The basic problem is CORRELATION CLUSTERING: Given a graph with real edge weights, partition the vertices into clusters to minimize the total absolute weight of cut positive edges and uncut negative edges. They consider the special case of a complete graph with each edge labeled with either $+1$ or $-1$, which is equivalent to CLUSTER EDITING. They give another proof that CLUSTER EDITING is *NP*-complete. The main focus is on approximation; they give a constant-factor approximation for minimizing disagreement (equivalent to the number of edit operations, as considered by Shamir et al. for CLUSTER DELETION), and a PTAS[1] for maximizing agreement (equivalent to the number of non-edited edges). The approximation for minimizing disagreement was improved to a factor-4 approximation by Charikar, Guruswami and Wirth [CGW03], who also show that CLUSTER EDITING is *APX*-hard with this objective.

From the more abstract view of graph modification problems, Leizhen Cai [Cai96] (also cf. [DF99]) considered the more general graph modification problem that allows edge deletions, edge additions, *and* vertex deletions. He showed that if the graph property is hereditary and can be characterized with a finite set of forbidden induced subgraphs, the problem is fixed-parameter

[1]Polynomial time approximation scheme, i.e., a scheme which, for any fixed $\varepsilon$, yields a polynomial time approximation within a $(1 + \varepsilon)$ factor. Running time may depend exponentially (or worse) on $1/\varepsilon$, though, making the resulting algorithms often impractical for small $\varepsilon$.

tractable. This result can be applied to CLUSTER EDITING and CLUSTER DELETION with $\{P_3\}$ as the forbidden set, and implies an $O(3^k \cdot |G|^4)$ time algorithm for both CLUSTER EDITING and CLUSTER DELETION. However, the resulting algorithms do not seem very suitable for practical implementations.

Kaplan, Shamir, and Tarjan [KST99] and Mahajan and Raman [MR99] also considered special cases of edge modification problems with particular emphasis on fixed-parameter tractability results.

### 2.2.2 A Parameterized Algorithm for CLUSTER EDITING

We will first show some simple properties of cluster graphs and the CLUSTER EDITING problem, and then present a very simple parameterized algorithm to solve CLUSTER EDITING based on these observations.

It is easy to observe that in CLUSTER EDITING for a graph $G$, it is never useful to connect two connected components. Therefore, if $G$ has more than one connected component, we can solve the problem independently on each connected component. This allows us in the following to assume that the input graph is connected.

Note further that it is possible to build a database which contains optimal solutions for all small graphs. Therefore, an algorithm can assume that every input instance has at least $c$ vertices, for a constant $c$.

The following characterization of cluster graphs is useful, since it allows deductions from only looking at small subgraphs:

**Lemma 2.1.** *[SST02] A graph is a cluster graph iff it contains no $P_3$, i.e., a path of three vertices, as induced subgraph.*[2]

In other words, cluster graphs can be characterized by the forbidden set $\{P_3\}$. Lemma 2.1 immediately leads to the following search tree algorithm, as illustrated in Fig. 2.1:

**Algorithm 2.1.** *Input: Graph $G$, parameter $k$*

- *If $G$ already is a cluster graph, we are done: report the solution and return.*
- *Otherwise, if $k \leq 0$, then we cannot find a solution in this branch of the search tree: return.*
- *Otherwise, find an induced $P_3$, i.e., three vertices $u, v, w \in V$ such that $\{u,v\} \in E$ and $\{u,w\} \in E$, but $\{v,w\} \notin E$ (they exist with Lemma 2.1; we call this a* conflict triple*). Recursively call the branching procedure on the following three instances consisting of graphs $G' = (V, E')$ with nonnegative integer $k'$ as specified below:*

[2]Note that Shamir et al. [SST02] write "$P_2$" for a path with three vertices, whereas we keep the notation commonly used in graph theory literature.

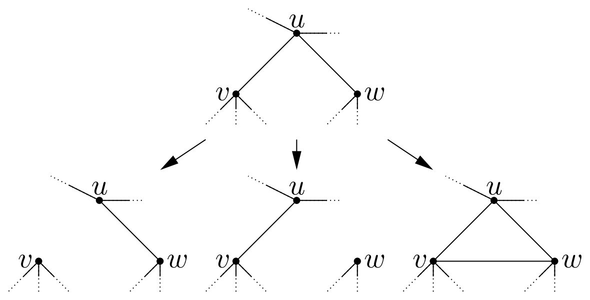

Figure 2.1: Simple branching for Cluster Editing

① $E' := E - \{u, v\}$ *and* $k' := k - 1$.

② $E' := E - \{u, w\}$ *and* $k' := k - 1$.

③ $E' := E + \{v, w\}$ *and* $k' := k - 1$.

This leads us to the following theorem:

**Theorem 2.1.** Cluster Editing *can be solved in* $O(3^k \cdot |E|)$ *time.*

*Proof.* The correctness of Algorithm 2.1 is obvious. Regarding running time, we note that the height of the search tree is bounded by $k$, since the parameter gets decreased by 1 in each layer. Since each internal node has 3 children, the search tree has $3^k$ leaves and $O(3^k)$ nodes.

In each node, we can in $O(|E|)$ time distinguish two cases by checking every edge:

- There is an an edge connecting two vertices $u$ and $v$ of unequal degrees with $\deg(u) > \deg(v)$. Then, clearly at least one neighbor $w$ of $u$ is not connected to $v$, and the three vertices $u$, $v$ and $w$ form a $P_3$. Finding such a neighbor takes $O(|E|)$ time.

- Otherwise, the graph consists of connected components where all vertices in a connected component have the same degree. Choose any vertex $u$ from a connected component that is not already a clique. Not all neighbors of $u$ can be pairwise connected by an edge, because otherwise $u$ and its neighbors would form a connected component that is a clique. Therefore, two neighbors $v$ and $w$ of $u$ exist that are not connected by an edge, and $u$, $v$ and $w$ form a $P_3$. If we check all pairs of neighbors of $u$, we will clearly have to examine less than $|E|$ pairs before finding one that is not connected by an edge.

In summary, we need $O(|E|)$ time per search tree node. □

Algorithm 2.1 can be improved in two ways:

- Reduce the overhead per node of the search tree;
- Prune the search tree with a more refined case distinction.

Both ways have been explored in previous works [GGHN03b]. In the remaining sections of this chapter, we will recapitulate the results and add a few remarks.

## 2.3 Annotations and Reductions

It turns out to be useful to consider an annotated version of CLUSTER EDITING, where the input graph has additional properties: each unordered pair of vertices $\{u, v\}$ can be marked as *immutable*. The meaning of this annotation depends on whether the graph contains the edge $\{u, v\}$:

- If $\{u, v\} \in E$ and $\{u, v\}$ is immutable, then the edge $\{u, v\}$ is called *permanent*. The algorithm then has to assume this edge will be part of any clustering solution, and must not delete it. With *marking permanent*, we denote the process of setting the immutable attribute for $\{u, v\}$, and adding $\{u, v\}$ to $E$ if it is not already there, decreasing the parameter $k$ by one in that case.
- If $\{u, v\} \notin E$ and $\{u, v\}$ is immutable, then $\{u, v\}$ is called *forbidden*. The algorithm then has to assume the edge $\{u, v\}$ will not be part of any clustering solution, and must not add it. With *marking forbidden*, we denote the process of setting the immutable attribute for $\{u, v\}$, and deleting $\{u, v\}$ from $E$ if it is present, decreasing the parameter $k$ by one in that case.

This annotation is already advantageous for the simple $3^k$ search tree: at each branch, the modified edge can be marked immutable, since changing it again cannot lead to an optimal solution. Therefore, in the following, we will assume that the considered CLUSTER EDITING instances are annotated, and that whenever edges are added or deleted, the immutable attribute is set for them.

The technique of introducing annotations to capture additional information acquired within the run of a search tree algorithm is well-known. For example, a search tree algorithm for the PLANAR DOMINATING SET problem annotates vertices as "white", meaning they are already dominated by another vertex [AFF$^+$01].

The annotations have the additional benefit of giving rise to the following reduction rules, as illustrated by Fig. 2.2:

**Rule 1.** If there are three pairwise distinct vertices $u, v, w$ in $V$ with $\{u, v\}$ permanent and $\{v, w\}$ permanent, then we can mark $\{u, w\}$ permanent.

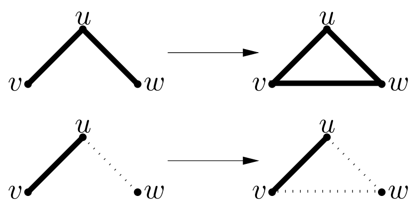

Figure 2.2: Reduction rules for CLUSTER EDITING. *Dotted lines* denote forbidden vertex pairs, *bold lines* denote permanent edges

**Rule 2.** If there are three pairwise distinct vertices $u, v, w$ in $V$ with $\{u, v\}$ permanent and $\{v, w\}$ forbidden, then we can mark $\{u, w\}$ forbidden.

*Proof.* The correctness of these rules is easy to see. □

## 2.4 Problem Kernel for CLUSTER EDITING

For parameterized problems, it is often possible to reduce a problem instance $I$ to an "equivalent" smaller instance $I'$ by applying *reduction rules.* A *reduction rule* replaces, in polynomial time, a given CLUSTER EDITING instance $(G, k)$ consisting of a graph $G$ and a nonnegative integer $k$ by a "simpler" instance $(G', k')$ with $k' \leq k$ such that $(G, k)$ has a solution iff $(G', k')$ has a solution, i.e., $G$ can be transformed into disjoint clusters by deleting/adding at most $k$ edges iff $G'$ can be transformed into disjoint clusters by deleting/adding at most $k'$ edges. An instance to which none of a given set of reduction rules applies is called *reduced* with respect to these rules.

By repeated application of reduction rules, for some problems it is possible to create an instance whose size is bounded by a function of the parameter $k$, independent of the original input size. When additionally interleaving search tree branching and reductions [NR00a], this makes the problem-size dependent factor in the $f(k)$ term go away, to be replaced with an additive term for the preprocessing. This is called *reduction to a problem kernel.*

We will now sketch the general scheme of a problem kernel reduction for CLUSTER EDITING.

**Rule 1** For every pair of vertices $u, v \in V$:

- If $u$ and $v$ have more than $k$ common neighbors, i.e., $|\{N(u) \cap N(v)\}| > k$, then the edge $\{u, v\}$ needs to belong to any clustering solution. We mark $\{u, v\}$ as permanent.
- If $u$ and $v$ have more than $k$ non-common neighbors, i.e., $|\{N(u) \triangle N(v)\}| > k$, then the edge $\{u, v\}$ cannot be part of a clustering solution. We mark $\{u, v\}$ as forbidden.

- If $u$ and $v$ have both more than $k$ common and more than $k$ non-common neighbors, then the given instance has no solution.

**Rule 2** Apply the reduction rules from Sect. 2.3.

**Rule 3** Delete the connected components which are cliques from the graph.

Applying these rules leads to the following theorem:

**Theorem 2.2.** CLUSTER EDITING *has a problem kernel which contains at most* $2(k^2+k)$ *vertices and at most* $2\binom{k+1}{2}k$ *edges. It can be found in* $O(|V|^3)$ *time.*

*Proof.* Because of Rule 3, none of the remaining connected components can be a clique. Therefore, there can be at most $k$ connected components, since each connected component requires at least one editing operation. With Rule 1, it can be shown that for each connected component, the number of vertices is limited by $2(k+1) \cdot k_i$ and the number of edges by $2\binom{k+1}{2}k_i$, where $k_i$ is the number of editing operation alloted to the $i$-th component. Since $\sum_i k_i = k$, the theorem is proven. $\square$

We refer to [GGHN03b] for the details of the proof and an analysis of the run time of the reduction rules.

Together with the simple search tree algorithm, we arrive at the following theorem:

**Theorem 2.3.** CLUSTER EDITING *can be solved in* $O(3^k + |V|^3)$ *time.*

## 2.5 Refined Search Tree for CLUSTER EDITING

The branching strategy from Sect. 2.2.2 can be improved as described in the following. We still identify a conflict triple of vertices, i.e., $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$. Based on a case distinction, we provide for every possible situation additional branching steps. The amortized analysis of the successive branching steps, then, yields the better worst-case bound on the running time.

We start with distinguishing three main situations that may apply when considering the conflict triple:

(C1) Vertices $v$ and $w$ do not share a common neighbor, i.e., $\nexists x \in V, x \neq u : \{v, x\} \in E$ and $\{w, x\} \in E$.

(C2) Vertices $v$ and $w$ have a common neighbor $x$ and $\{u, x\} \in E$.

(C3) Vertices $v$ and $w$ have a common neighbor $x$ and $\{u, x\} \notin E$.

Regarding case (C1), we show in the following lemma that, here, a branching into two cases suffices.

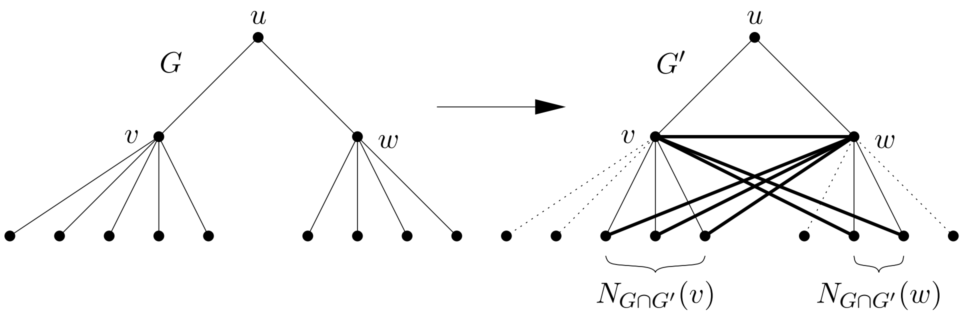

Figure 2.3: Illustration of Lemma 2.2. Here, $G$ is the given graph and $G'$ is a clustering solution of $G$ obtained after adding the edge $\{v, w\}$. The *dashed lines* denote the edges being deleted to transform $G$ into $G'$, and the *bold lines* denote the edges being added. Observe that the drawing only shows that parts of the graphs (in particular, edges) which are relevant for our argumentation

**Lemma 2.2.** *Given a graph $G = (V, E)$, a nonnegative integer $k$ and $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$. If $v$ and $w$ do not share a common neighbor, then the branching case of adding the edge $\{v, w\}$ cannot yield a better solution than both the cases of deleting $\{u, v\}$ or $\{u, w\}$. Therefore, it can be omitted in the branching.*

*Proof.* Consider a clustering solution $G'$ for $G$ where we did add $\{v, w\}$ (see Fig. 2.3). We use $N_{G\cap G'}(v)$ to denote the set of vertices which are neighbors of $v$ in both $G$ and in $G'$. Without loss of generality, assume that $|N_{G\cap G'}(w)| \leq |N_{G\cap G'}(v)|$. We then construct a new graph $G''$ from $G'$ by deleting all edges adjacent to $w$. It is clear that $G''$ is also a clustering solution for $G$. We compare the cost of the transformation $G \to G''$ to that of the transformation $G \to G'$:

- $-1$ for not adding $\{v, w\}$,
- $+1$ for deleting $\{u, w\}$,
- $-|N_{G\cap G'}(v)|$ for not adding all edges $\{w, x\}$, $x \in N_{G\cap G'}(v)$,
- $+|N_{G\cap G'}(w)|$ for deleting all edges $\{w, x\}$, $x \in N_{G\cap G'}(w)$.

Herein, we omitted possible vertices which are neighbors of $w$ in $G'$ but not neighbors of $w$ in $G$ because they would only increase the cost of transformation $G \to G'$.

In summary, the cost of $G \to G''$ is not higher than the cost of $G \to G'$, i.e., we do not need more edge additions and deletions to obtain $G''$ from $G$ than to obtain $G'$ from $G$. $\square$

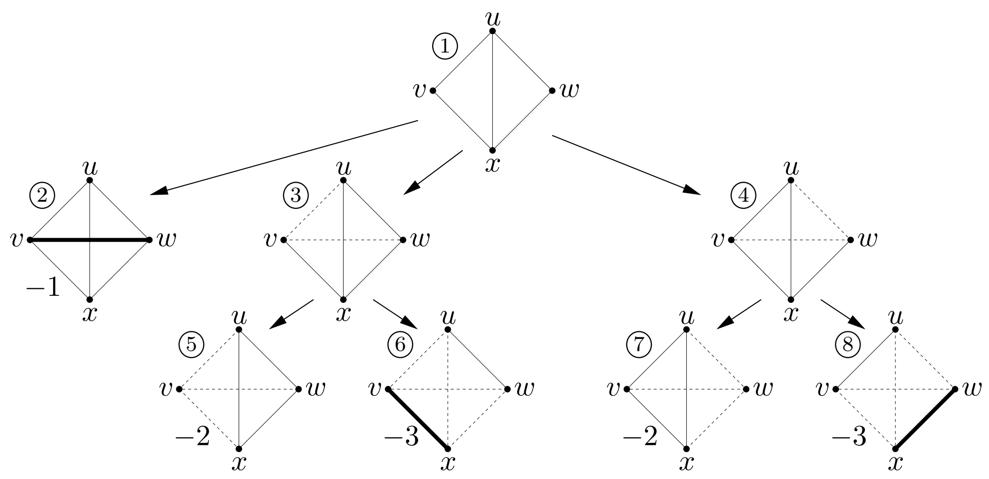

Figure 2.4: Branching for case (C2). *Bold lines* denote permanent, *dashed lines* forbidden edges

Lemma 2.2 shows that in case (C1) a branching into two cases is sufficient, namely to recursively consider graphs $G_1 = (V, E - \{u, v\})$ and $G_2 = (V, E - \{u, w\})$, each time decreasing the parameter value by one.

For case (C2), we change the order of the basic branching. In the first branch, we add edge $\{v, w\}$. In the second and third branches, we delete edges $\{u, v\}$ and $\{u, w\}$, as illustrated by Fig. 2.4.

- Add $\{v, w\}$ as labeled by ② in Fig. 2.4. The cost of this branch is 1.
- Mark $\{v, w\}$ as forbidden and delete $\{u, v\}$, as labeled by ③. This creates the new conflict triple $(x, u, v)$. To resolve this conflict, we make a second branching. Since adding $\{u, v\}$ is forbidden, there are only two branches to consider:
  - Delete $\{v, x\}$, as labeled by ⑤. The cost is 2.
  - Mark $\{v, x\}$ as permanent and delete $\{u, x\}$. With reduction rule 2 from Sect. 2.3, we then delete $\{x, w\}$, too, as labeled by ⑥. The cost is 3.
- Mark $\{v, w\}$ as forbidden and delete $\{u, w\}$ (④). This case is symmetric to the previous one, so we have two branches with costs 2 and 3, respectively.

In summary, the branching vector for case (C2) is $(1, 23, 2, 3)$.
For case (C3), we perform a branching as illustrated by Fig. 2.5:

- Delete $\{u, v\}$, as labeled by ②. The cost of this branch is 1.

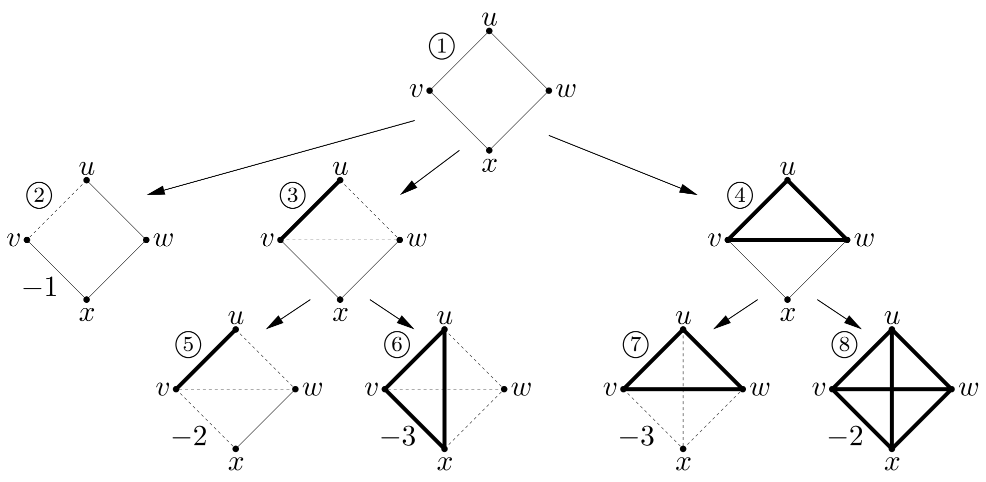

Figure 2.5: Branching for case (C3)

- Mark $\{u, v\}$ as permanent and delete $\{u, w\}$, as labeled by ③. With Rule 2, we can additionally mark $\{v, w\}$ as forbidden. We then identify a new conflict triple $(v, u, x.)$ Not being allowed to delete $\{v, u\}$, we can make a 2-branching to resolve the conflict:
  - Delete $\{v, x\}$, as labeled by ⑤. The cost is 2.
  - Mark $\{v, x\}$ as permanent. This implies $\{x, u\}$ needs to be added and $\{x, w\}$ to be deleted, as labeled by ⑥. The cost is 3.
- Mark $\{u, v\}$ and $\{u, w\}$ as permanent and add $\{v, w\}$, as labeled by ④. Vertices $(w, u, x)$ form a conflict triple. To solve this conflict without deleting $\{w, u\}$, we make a 2-branching:
  - Delete $\{w, x\}$ as labeled by ⑦. We then also need to delete $\{x, v\}$. The cost is 3. Additionally, we can mark $\{x, u\}$ as forbidden.
  - Add $\{u, x\}$, as labeled by ⑧. The cost is 2. Additionally, we can mark $\{x, u\}$ and $\{x, v\}$ as permanent.

It follows that the branching vector for case (C3) is $(1, 2, 3, 3, 2)$.

In summary, this leads to a refinement of the branching with a worst-case branching vector of $(1, 2, 2, 3, 3)$, yielding the branching number 2.27. Since the recursive algorithm stops whenever the parameter value has reached 0 or below, we obtain a search tree size of $O(2.27^k)$. This results in the following theorem:

**Theorem 2.4.** Cluster Editing *can be solved in* $O(2.27^k + |V|^3)$ *time.*

## 2.6 Heuristic Improvements

The following rules do not affect the worst-case time complexity, since there is no guarantee that any of them ever applies; however, they might be useful for practical implementation, since they are fairly cheap and can help reduce the size of the search tree substantially if they do apply.

### Reduction Rules

In some cases, no branching is needed, and an instance $G$ with parameter $k$ can be directly replaced with a simplified instance $G'$ with parameter $k'$. The correctness of the following rules can be easily seen with the above branching rules and symmetry arguments.

Let $u, v, w, x, y$ be distinct vertices.

- If $\deg(u) = \deg(v) = 1$ and $N(u) = N(v) = \{w\}$, then delete $\{u, w\}$ and set $k' := k - 1$.
- If $\deg(u) = 1, \deg(v) = 2$, $N(u) = \{v\}$ and $N(v) = \{u, w\}$, then delete $\{v, w\}$ and set $k' := k - 1$.
- If $\deg(u) = 2, \deg(v) = \deg(w) = 3$ and $N(u) = \{v, w\}, N(v) = \{u, w, x\}, N(w) = \{u, v, y\}$, then delete $\{v, x\}$ and $\{w, y\}$ and set $k' := k - 2$.

### Branching Rules

For some local substructures, special branchings can be identified that have noticeably better branching vectors than the "normal" branching. Therefore, it is beneficial to execute these branchings if possible and only fall back to the general search tree when they are no longer applicable.

- If $\{u, v\} \in E$ and $u$ and $v$ do not have a common neighbor, branch into two cases: either delete $\{u, v\}$, or delete all edges adjacent to $u$ or $v$ except $\{u, v\}$. This rule is proven and utilized further in Sect. 4.1.

### Bail-Out Rules

Some branches in the search tree need not be followed, since either they cannot lead to a solution, or because it is known that for any solution they might lead to, we find another solution which is at least as good in another branch.

- Let $G_0$ be the original input graph and let $G$ be the graph in the current state of the algorithm. If $G$ contains a vertex $v$ with $\deg_G(v) \geq 2\deg_{G_0}(v)$ then the current branch of the search tree can be omitted, since we can be certain to find an optimal solution in another branch of the search tree. This rule is correct as can be seen as follows: for any possible clustering solution $G'$ of $G$, we can construct another clustering solution $G''$ by removing all edges adjacent to $v$ in $G'$. Clearly, the cost of transforming $G_0$ to $G''$ is not higher than the cost of transforming $G_0$ to $G'$.

# Chapter 3

# Automated Discovery of Branching Rules

The branching algorithm given in the previous chapter for CLUSTER EDITING follows the following basic scheme:

(1) Find a subgraph induced by three vertices that is a $P_3$, i.e., a conflict triple.

(2) Extend the subgraph according to given rules by adding vertices from its neighborhood.

(3) Look up a branching rule for the resulting subgraph in a dictionary containing an entry for every possible subgraph, and branch into the cases determined by this branching rule.

This recursive procedure realizes the following very general search tree algorithm scheme, which requires the possibility to consider only a "small part" of the input, a *window*, and draw conclusions about the solution from this window. We can sketch this scheme as follows.

**General search tree algorithm scheme.**

(1) Find an initial window in the input.

(2) Expand the window according to a given *expansion rule*, usually by adding information that is "local".

(3) Look up a branching rule for the resulting window and branch into the cases determined by the rule. This requires a dictionary containing branching rules for all windows which might be produced in step (2).

To ensure termination and make complexity analysis possible, each case of a branching rule must decrement a problem parameter $k$, which can either be a parameter in the sense of parameterized complexity (Sect. 1.3.2), or simply the problem size. For a window, the amount the parameter is decreased compared to to the initial input is referred to as the *cost*.

Trying to define the terms "window" and "expand locally" rigorously, one would most likely end up with concepts too general for practical usefulness; therefore, we will only illustrate them with a few examples:

- For graph modification problems, an obvious (and well-tried) choice for windows is induced subgraphs. Expanding locally means adding vertices adjacent to the subgraph.
- For satisfiability problems, the window could be a small set of clauses. It could be expanded e.g. by adding a clause that contains a variable which is already considered.
- For job scheduling on a single machine with precedence constraints, a window could be a subset of the jobs. Expanding it means adding jobs with certain relations to jobs already considered.

While the examples suggest a wide applicability of the scheme, there are problems where it is not obvious how it could be applied, for example Hamiltonian Cycle (given a graph of $n$ vertices, decide whether it contains a spanning cycle): a small induced subgraph of the input does not seem to allow any conclusions about a possible solution, for example whether a particular edge is part of the spanning cycle.

The performance of an algorithm that follows this scheme depends on the worst case of the branching numbers of the stored branching rules in step (3), since, for a worst-case analysis, one has to assume that the expansion in step (2) always yields the most adversarial window. Therefore, there are two key points in finding an efficient search tree algorithm:

- The choice of expansion rules employed in step (2) of the general search tree scheme. The goal is to find an expansion rule where the set of possible expanded window when considering all possible inputs is advantageous. Presumably, the larger this set is, the better, since then each element gives more information to its branching rule. This is going to be the topic of Chap. 4.
- Find a good branching rule for a given window, i.e., to build the branching rule dictionary used in step (3) of the general search tree scheme. In this chapter, we will develop a method for this which can even be considered optimal under certain criteria.

We will first present an initial automated approach for finding good algorithms following this scheme for Cluster Editing, and then formalize and generalize the method.

## 3.1 Simplified Branching for Cluster Editing

To ease automation, we start by simplifying and systematizing the elements of the "manually" found algorithm presented in Sect. 2.5.

Firstly, let us consider the case enumeration part. In the manually found algorithm, we gave a branching rule only for two graphs of size 4; the others were "discussed away" by arguing that they cannot occur in instances reduced with respect to a certain preprocessing (Lemma 2.2). For the automation approach, we simply consider all possible graphs resulting from adding an arbitrary vertex to the conflict triple, i.e., all size-4 graphs containing a $P_3$.

Secondly, we consider the branching rules for each case. Looking at Fig. 2.4, we see that we arrived at the final cases of the branching rule by a hierarchical case distinction, which we call a *branching tree*. The root of a branching tree is the window for which it defines a branching rule, and each leaf marks a branching case. The step from an internal node to its children is called a *sub-branching*; for example, the first sub-branching into three cases in Fig. 2.4 is based on the trivial 3-branching on a conflict triple.

The technique of sub-branching seems appropriate for automation, since one only needs to prove correctness of the sub-branching rules to show the correctness of a branching rule. All branching rules we are going to consider with our automated framework are based on a branching tree, therefore we will use the two terms mostly interchangeably.

In the manually found algorithm, within both branching rules (Fig. 2.4 and 2.5), we used complicated reasoning to justify the completeness of the sub-branchings. For automation purposes, we need to find simpler and more general rules.

It turns out that for CLUSTER EDITING, one can already get branching rules as good as those of Sect. 2.5 (case (C2) and (C3)) based on only two principles for the sub-branchings:

- A single kind of sub-branching: For a pair of vertices $u$ and $v$ for which $\{u, v\}$ is not annotated as immutable yet, branch into two cases, one where $\{u, v\}$ is marked permanent, and one where $\{u, v\}$ is marked forbidden. The completeness of this case distinction is obvious.

- The application of the reduction rules from Sect. 2.3.

Figure 3.1 shows a branching rule for an exemplary size-4 graph based on this scheme, which was generated automatically by our framework. It has the same branching vector as the carefully hand-crafted rule for this graph in Fig. 2.4. In fact, one can see a close correspondence between them; the initial sub-branching into three cases in Fig. 2.4 is expressed in the simplified branching of Fig. 3.1 by introducing an extra intermediary node.

The branching trees we are looking for are already completely defined by giving its structure as a binary tree, and stating in each node which edge to

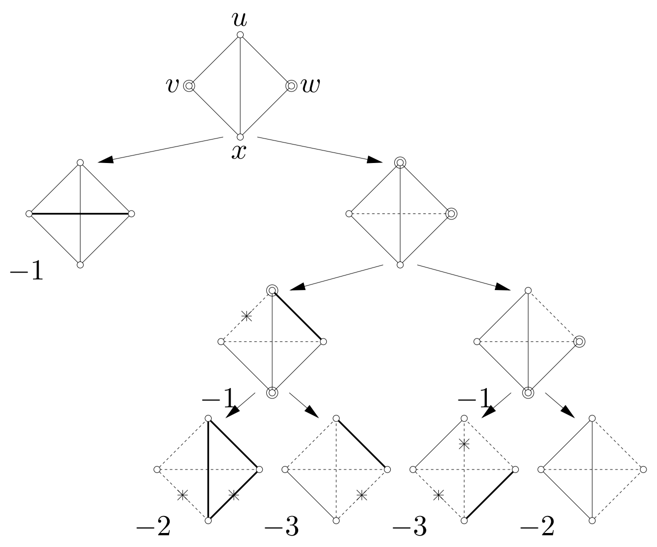

Figure 3.1: Branching tree for a Cluster Editing window using only subbranching on vertex pairs (*double circles*), and applications of the reduction rules (*asterisks*). The *numbers* next to the windows state the change of the problem parameter $k$

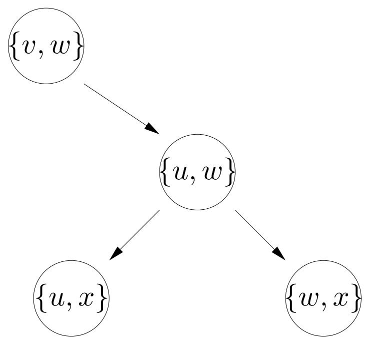

Figure 3.2: Abstract characterization (branching tree) of the branching rule from Fig. 3.1

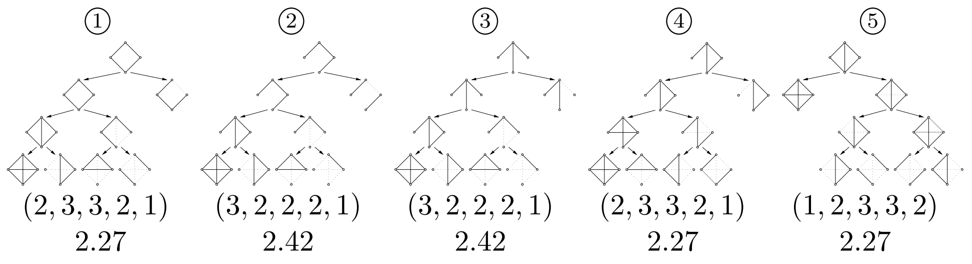

Figure 3.3: Sketch of the optimal branching rules for the 5 windows of size 4 containing a $P_3$, along with their branching vectors and branching numbers

| Size | Graphs | Trees |
|---:|---:|---:|
| 4 | 5 | 720 |
| 5 | 20 | 3.628.800 |
| 6 | 111 | 1.307.674.368.000 |

Table 3.1: Number of non-isomorphic size-$n$ graphs containing a $P_3$, and number of possible branching trees for each case (rough estimate)

branch on. Figure 3.2 shows this abstracted view of the branching tree from Fig. 3.1.

Clearly there is only a finite amount of such branching rules; we can try them all and choose the one with the best resulting branching vector. Figure 3.3 sketches the optimal branching rules for each of the 5 possible graphs that can be generated by extending the $P_3$ by one vertex. The worst cases are cases ② and ③ with a branching number of 2.42 each.

Comparing this to the manually designed algorithm, we note that for the two graphs for which we manually determined a branching, the automated method finds very similar branchings with identical branching vectors. However, since the automated analysis considers more cases, namely all size-4 graphs containing a $P_3$, the resulting search tree still has a worse time complexity.

Therefore, the next step is obviously to utilize the brute force of the machine and examine all cases where *two*, or even more vertices are added to the conflict triple. Unfortunately, this implies a very steep increase in computational cost, since (a) there are more cases to distinguish, and (b) more branching trees to try for each case, as illustrated by Table 3.1. We will estimate the number of branching trees only very roughly. We simply give the number of complete binary trees uniquely labeled with vertex pairs for size-$n$ graphs: $((n \cdot (n-1))/2)!$. Thereby, we ignore both that truncated trees are also allowed (which increases the actual number of possible branching trees), and that many labellings must be dismissed because one can only branch on edges that have not been marked as immutable by the reduction rules (which decreases the number of possible branching trees). Because of the application of reduction rules, the actual number of branching trees also depends on the window considered.

The figures indicate that it is not feasible to simply enumerate all possible branching rules for an instance; the space of branching rules needs to be searched with a more sophisticated method, which will be presented in the following section.

## 3.2 Finding Optimal Branching Rules

We will now give an algorithm that, given an expanded window from step (2) of the general algorithm scheme presented in the previous section, finds an optimal branching rule for step (3), i.e., one which has the lowest branching number of all branching rules based on this scheme. For this, we need several problem-specific data structures and functions, which are first given in a general way and will then be made more concrete with the example of Cluster Editing:

1. A way to represent a window of the input.
2. A function that yields for a window the set of possible *branching objects*. Branching objects mark each node in the sub-branching representation of a branching rule, i.e., the branching tree.
3. A function which yields for a window and a branching object a set of new windows.

For Cluster Editing, we made the following choices:

1. A window is an induced subgraph (with annotations).
2. Branching objects are vertex pairs (see Fig. 3.2). A vertex pair is only a legal branching object if it is not marked immutable.
3. The set of cases in a sub-branching always consists of two graphs, with the vertex pair marked permanent in the first one, and marked forbidden in the second one. Both graphs will additionally be reduced according to the reduction rules from Sect. 2.3.

To make the explanations more understandable, we will use the terminology of Cluster Editing, but the algorithm is trivially generalizable to other problems.

To compute a search tree branching rule, we, again, use a search tree to explore the space of possible branching rules. This search tree is referred to as *meta search tree*.

An obvious approach to explore the search space is *Divide&Conquer*. For each vertex pair $\{u, v\}$, we consider all branching rules that branch on this

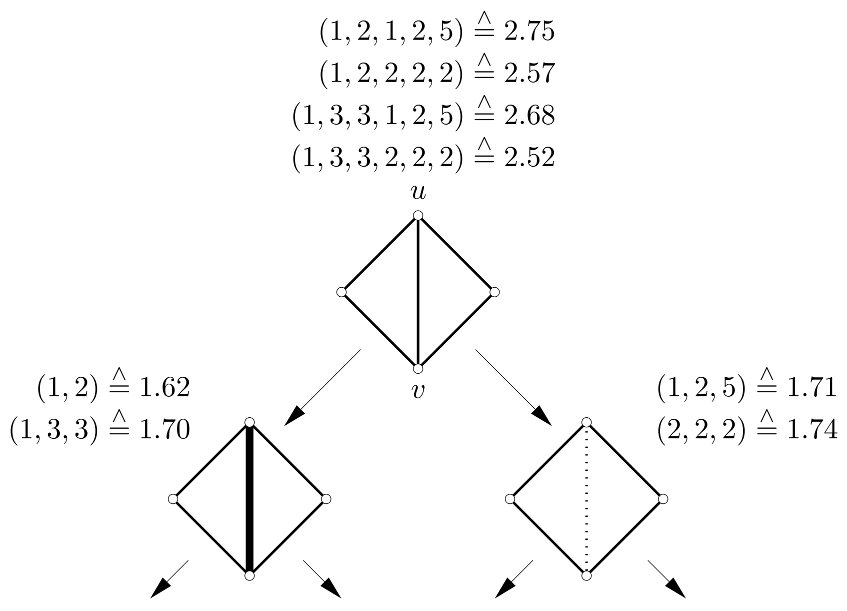
$(1,2,1,2,5) \triangleq 2.75$
$(1,2,2,2,2) \triangleq 2.57$
$(1,3,3,1,2,5) \triangleq 2.68$
$(1,3,3,2,2,2) \triangleq 2.52$
$u$
$v$
$(1,2) \triangleq 1.62$
$(1,3,3) \triangleq 1.70$
$(1,2,5) \triangleq 1.71$
$(2,2,2) \triangleq 1.74$

Figure 3.4: Fragment of a meta search tree which shows that passing only the optimal branching vectors does not suffice

pair as first sub-branching. We mark this pair permanent and recursively calculate an optimal branching rule for this instance; likewise for the graph with $\{u, v\}$ marked forbidden. Then we can generate the branching tree by joining the two returned optimal branching trees with a node that denotes a branch on $\{u, v\}$. Finally, determine the best vertex pair to branch on initially by comparing all branching vectors.

Unfortunately, it turns out that this method does not yield optimal branching rules; this is because of the counter-intuitive fact that joining two optimal branching trees does not necessarily yield an optimal branching tree. Figure 3.4 illustrates this with an example. Suppose the search in the left branch of the meta search tree returns two branching trees, one branching into two cases with $k$ reduced by 1 and 2, and the other branching into three cases with $k$ reduced by 1, 3, and 3. The branching vector $(1, 2)$ has a better branching number, so all other search trees would be discarded (in this case only one). On the right part of the meta search tree, also two search trees are found, the better one having the branching vector $(1, 2, 5)$. Joining the two optimal search trees produces a branching tree with branching vector $(1, 2, 1, 2, 5)$ and branching number 2.75. However, this is not the best possible branching; in fact, joining the *worst* cases of both sub-problems leads to the optimal branching vector of $(1, 3, 3, 2, 2, 2)$ with a considerably better branching number of 2.52.

Therefore, at each node of the meta search tree, a whole *set* of search trees has to be returned. Only at the root of the meta search tree, we can then compare their branching numbers and select an optimal one. It is obvious that there is a combinatorial explosion in the size of the search tree sets; indeed, for the practical implementation, this turned out to be a major computational bottleneck. We conceive that eventually one will have to employ heuristics here to make the search feasible for larger problems.

### 3.2.1 Algorithm Description

We will now define our meta search tree procedure more formally. Our central reference point in this section is the function *find_branching* given in Fig. 3.5.

We use several data types:

*Window:* A window represents the current subgraph including annotations. Also stored is the number of modifications already applied to the input window (*cost*).

*Branch_obj:* A branching object (vertex pair in the case of CLUSTER EDITING).

*Problem:* A structure containing two methods: *branching_objs*, which returns a list of all possible branching objects for a window; and *branches*, which returns a pair of windows with the possible branching cases of the branching object (for CLUSTER EDITING, the graph with the vertex pair marked permanent or forbidden, respectively).[1]

*Btree_set:* A set of branching trees, each represented as binary tree where internal nodes are labeled with branching objects (see Fig. 3.2). Leaves contain the cost of the corresponding window. Provided are methods to create and merge such sets.

The function *find_branching* takes two parameters: a window *window* and a problem structure *problem* that provides the problem-specific functions mentioned at the beginning of Sect. 3.2: identifying branching objects (*problem.branching_objs*), and applying sub-branching to a window given a branching object to yield new windows (*problem.branches*).

The result of *find_branching* is a set of branching trees. It is guaranteed that among the returned branching trees, there is an optimal one, since (in the unrefined algorithm) it contains all possible branching trees. Therefore, one can get an optimal branching tree for a window by passing it to *find_branching* and selecting the branching tree with the lowest branching number from the resulting set.

[1]Other problems can have more than two branching cases, or a variable number of cases; taking this into account is straightforward, but omitted here for clarity.

```
let find_branching problem window =
  List.fold                                                         (* 2 *)
    (fun btree_set branch_obj →
      let win1, win2 = problem.branches window branch_obj in        (* 5 *)
      let btree_set1 = find_branching problem win1 in               (* 6 *)
      let btree_set2 = find_branching problem win2                  (* 6 *)
      in
        Btree_set.add                                               (* 8 *)
          btree_set
          (Btree_set.merge btree_set1 btree_set2 branch_obj))       (* 7 *)
    (if Window.cost window = 0                                      (* 4 *)
      then Btree_set.empty
      else Btree_set.make (Window.cost window))                     (* 3 *)
    (problem.branching_objs window)                                 (* 1 *)
```

Figure 3.5: Pseudo-Code for the meta search tree

Note that a branching vector must not contain a zero, since that corresponds to a branching tree which contains a leaf where the parameter is not decreased at all; this would represent a non-terminating search tree. Therefore, to get a useful result, one has to ensure that there is at least one valid search tree in the space defined by the problem and the employed way of sub-branching. The easiest way to do this is to have a branching function which only returns windows with decreased parameter. The branching function we chose for Cluster Editing does not have this property; only one of the two branches will decrease the parameter. It is easy to see, however, that if the input contains a $P_3$, the application of the reduction rules will lead to at least one valid search tree.

The process flow of *find_branching* is as follows (see Fig. 3.5):

The problem-specific function *problem.branching_objs* is called to generate a list of possible branching objects (1). For Cluster Editing, it contains a list of all vertex pairs that are not marked immutable. This list is folded (2) to get the resulting search tree set. The starting point is a one-element set containing a search tree that consists only of a leaf, representing no further branching (3). As explained in the previous paragraph, this is only a legal option if the parameter has already been decreased for this window (4). For each branching object, the two modified windows are retrieved (5), and a set of branching trees is calculated recursively for each of them (6). From the resulting sets, each possible joining is generated, with the current branching object as root (7) and added to the existing set (8).

**Correctness.** The algorithm returns *all* possible branching trees for the input window; therefore, it will also contain an optimal one.

## 3.3 Optimizations

It is clear that the joining of branching trees in each meta search tree node leads to a combinatorial explosion in the size of the branching tree sets; this is already the case with Cluster Editing, where only two sets are merged, but it gets even worse when there are more than two branching cases in a sub-branching to consider.

We will present two techniques to speed up the meta search tree procedure.

**Transposition tables.** Within the search space of the meta search tree, we often encounter *transpositions*, i.e., identical subproblems reached by distinct paths. For example, with Cluster Editing, the meta search tree will try to branch on vertex pair $\{a, b\}$ first and then on vertex pair $\{c, d\}$ for four distinct vertices $a, b, c, d$. Then, at a later stage, it will also examine branching trees that branch on $\{c, d\}$ first and then on $\{a, b\}$. This obviously leads to identical graphs, and a potentially very large search space will be researched. This problem is well known from state space search; the simplest technique to mitigate it is to employ a *transposition table*, which is a simple associative data structure that stores for each encountered subproblem the result.

While transposition tables are conceptually simple and can avoid *all* transpositions, their application is often not feasible due to excessive memory requirements. In our test cases, however, we could manage to keep all occurring subproblems in memory, and, thus, did not have to revert to more sophisticated transposition avoidance schemes. This might be different when applying the meta search tree to other problems.

**Pruning branching tree sets.** As elaborated in Sect. 3.2, one cannot remove a branching tree from a set of branching trees simply because it has a worse branching vector than another one from the same set, since joining with other branching trees can change this. However, the following observation allows to at least remove some of the branching trees.

Consider a branching vector as a multi-set, i.e., identical elements may occur several times but the order of the elements plays no role. Then, comparing two branching vectors $b_1$ and $b_2$, we say that $b_2$ is *subsumed by* $b_1$ if there is an injective mapping $f$ of elements from $b_1$ onto elements from $b_2$ such that for every $x \in b_1$ it holds that $x \geq f(x)$. Then, if one branching vector is subsumed by another one from the same set, the subsumed one can be discarded from further consideration because the other one always leads to better solution no matter what we concatenate to it.

Unfortunately, for branching vectors which are incomparable with respect to subsumption we do not see a useful simplification rule. Thus, we have to compare all remaining pairs of branching vectors. Since the corresponding

| $n$ | Cases | Search tree | Time |
|---|---|---|---|
| 4 | 5 | $O(2.42^k)$ | 0.01 sec |
| 5 | 20 | $O(2.27^k)$ | 2 sec |
| 6 | 111 | $O(2.16^k)$ | 9 days |

Table 3.2: Results for Cluster Editing from extending a $P_3$ up to a size-$n$ graph and finding an optimal branching rule for each resulting case

characteristic polynomials strongly "oscillate", we consider any result to improve this as a challenging task. It is conceivable, however, that heuristics for selecting concatenation candidates might work well.

## 3.4 Results for Cluster Editing

With the optimizations mentioned in the previous chapter, our framework is able to not only find branching rules for graphs of size 4, but also of size 5 and 6, while still finding optimal solutions. Table 3.2 shows the worst case sizes of the resulting search trees. Already when considering size-5 graphs, the complexity matches that of the elaborated "manually" found algorithm from Theorem 2.4; when examining size-6 graphs, one gets a considerable improvement. The running time to find these optimal branching rules, however, is considerable; most of it is not due to the increased number of nodes in the meta search tree, but due to the increased size of branching tree sets to be merged in each node.

## 3.5 Generalization for Other Graph Modification Problems

We will now demonstrate how one can apply the algorithm for finding branching trees to any graph modification problem where the target property can be characterized by a finite set of forbidden induced subgraphs; we will call this class of problems GMP-FS. This class covers many well known and important graph classes (see [BLS99, Sect. 7.1] for a survey). We will sometimes call the forbidden induced subgraphs *conflict graphs.*

First, we introduce an "immutable" annotation as described in Sect. 2.3. As with Cluster Editing, we will from now on only consider the annotated problem version.

Then, for each forbidden subgraph $F = (V_F, E_F)$, devise reduction rules as follows:

If a subset of the vertices of the current windows induces either $F$ or $(V_F, E_F \triangle \{\{u, v\}\})$ (i.e., $F$ with one vertex pair $\{u, v\}$ toggled), and all pairs of these vertices except $\{u, v\}$ are marked immutable, then we can

| Graph Class | Forbidden set |
|---|---|
| Cluster graphs | $\{P_3\}$ |
| Triangle-free graphs | $\{C_3\}$ |
| Unit interval graphs | $\{K_{1,3}\}$ |
| Cographs | $\{P_4\}$ |
| Trivially perfect graphs | $\{P_4, C_4\}$ |
| Split graphs | $\{2K_2, C_4, C_5\}$ |

Table 3.3: Some graph classes with a characterization by a set of forbidden induced subgraphs

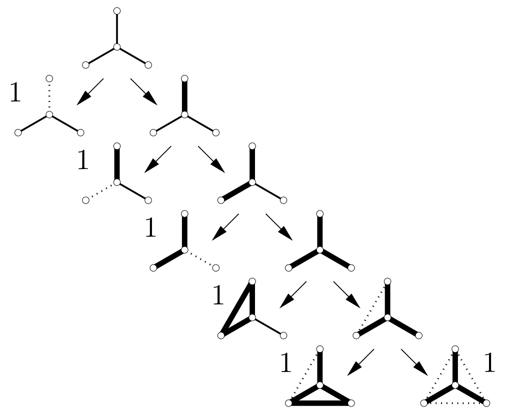

Figure 3.6: The trivial $6^k$-branching for Claw Editing, as discovered by our framework

- mark $\{u, v\}$ forbidden if $\{u, v\} \in E_F$;
- or mark $\{u, v\}$ permanent if $\{u, v\} \notin E_F$.

It is easy to see that instantiating this rule for Cluster Editing will result in the reduction rules from Sect. 2.3.

As with Cluster Editing, the sub-branchings always branch into two cases, marking a vertex pair permanent or forbidden, respectively.

The modifications needed to also cover edge deletion and vertex deletion problems are straightforward.

Conceivably, it would not be very hard to extend our framework to handle GMP-FS problems fully automatically in the sense that only the set of forbidden subgraphs needs to be given (see also Sect. 6.4 for more information on what is needed to add a problem to our framework).

The finite forbidden subset characterization covers a large class of graph modification problems, some of which are enumerated in Table 3.3. For all of them, a simple search tree algorithm can be given, analogous to that of Algorithm 2.1: Find a forbidden subgraph in the input, and branch into several cases, where in each case a different vertex pair is edited to destroy the conflict graph.

For a forbidden subgraph of $s$ vertices, we have $s \cdot (s-1)/2$ vertex pairs, and therefore this results in a search tree size of $O((s \cdot (s-1)/2)^k)$. The automated branching rule finder will "rediscover" this trivial branching by generating a binary tree of sub-branchings which edits each vertex pair of the conflict graph in one branch (which is then no longer followed, and has cost 1), and makes it immutable in the other branch, till finally only one non-immutable vertex pair is left and a reduction rule applies. Figure 3.6 demonstrates this for the Claw Editing problem, which has a claw as forbidden subgraph.

How well does this scheme work for problems other than Cluster Editing? One important reason it works so well for Cluster Editing and related problems with the $P_3$ as forbidden induced subgraph is the following property:

**Lemma 3.1.** *A connected graph with $n$ vertices that contains at least one $P_3$, contains at least $n-2$ $P_3$s.*

*Proof.* Structural induction starting from the $P_3$. □

This property seems very advantageous for our branching rule finder. An intuition for this can be given as follows: The gain over the trivial branching effectively comes from the application of the reduction rules. If an edge is annotated immutable by a sub-branching, and it is part of several conflict triples, the chances that additional sub-branchings will create an opportunity for a reduction increase.

Unfortunately, the applicability for GMP-FS problems alone does not guarantee getting an algorithm that is better than the trivial search tree. We found that for some GMP-FS problems, our framework, without further refinements, cannot find any nontrivial rules for windows that contain only one instance of the forbidden subgraph.

In the next chapter, we will demonstrate a technique to overcome this weakness. With the improvements from that chapter, we will present nontrivial results for several graph modification problems in Chap. 5.

Chapter 4

# Automated Discovery of Case Distinctions

In the previous chapter, we assumed a simplistic method of determining the set of expanded windows for each of which we invoke the optimal branching rule finder: for Cluster Editing, generate all possible size-$n$ graphs containing a $P_3$, or, for general problems, generate all possible windows which are expansions of the initial window up to a certain size.

There are two drawbacks to this naïve approach:

- Often, several cases of expanded windows can be combined into a single, more general case without worsening the branching number of the corresponding branching rule, thereby saving time in generating the rules, and eventually yielding a simpler algorithm.
- Since the total time complexity is determined by the worst case of the branching rules, just distinguishing more and more cases often does not help; for example for Triangle Deletion, there are windows of arbitrary size for which no nontrivial branching rule can be found based on our branching tree scheme.

The basic principle to improve upon the simplistic method is to devise *specific* expansion rules aimed at creating a set of possible expanded windows for which efficient branching rules can be found, while still covering all possible inputs.

**Example.** Given a graph problem with the property that the input graphs have a minimum vertex degree of three. Naïvely, we generate a set of windows covering any input by successively adding vertices to an initially empty window, up to size 4, yielding every possible connected graph of size 4. Assume that the window with the worst branching rule is the $P_4$. Then we can improve the result by requiring that the $P_3$ gets specifically expanded

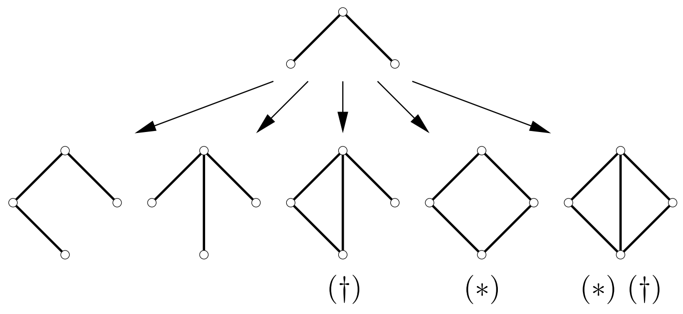

Figure 4.1: Expansion of the $P_3$. The trivial expansion of adding an arbitrary vertex yields the five windows depicted in the lower part. When applying the specific expansion based on Sect. 2.5 (which adds the common neighbor of two $P_3$ endpoints), we get only the two windows marked with (∗); when applying the specific expansion from Lemma 4.2 for either edge of the $P_3$, we get only the two windows marked with (†)

by adding another neighbor of its middle vertex, which must exist by the special property of the input. That way, a $P_4$ can never be generated, and we can improve the overall branching number. □

Even if the input is not structured in any useful way, like for Cluster Editing where the input is a general graph, this idea can still be applied: We identify "trivial" instances of the problem, for which we devise special branching rules, and assume them to be eliminated before applying the window expansion, thereby creating invariants which can be utilized to restrict the set of generated cases.

When analyzing the performance of a search tree algorithm which employs this technique, one has to keep in mind that its worst-case search tree size is determined by the worst case of the trivial branchings and the branching using expansion and case distinction. Therefore, in devising trivial branchings, one needs to find a trade-off between the amount of additional structure they provide, and their branching number.

In fact, we have already seen in Chap. 2 an application of trivial branchings: In the refined search tree algorithm for Cluster Editing, we identified the case in which the two endpoints of a $P_3$ do not have a common neighbor as trivial (Sect. 2.5). This gives us a new invariant about the structure of the input: it is not an arbitrary graph anymore, but we may assume that for each such pair of endpoints, we can find an additional common neighbor. Therefore, we can expand a window not only by adding an arbitrary vertex from the neighborhood, but also selectively by adding this common neighbor of the endpoints of a $P_3$ within the window, if it is not already represented in the window. With this expansion method, we only get two possible expanded windows when expanding the $P_3$ (Fig. 2.4 and 2.5), compared to five when adding an arbitrary vertex (see Fig. 4.1).

We will demonstrate the power of this approach first by giving a problem-specific expansion rule for CLUSTER EDITING in the next section, and then a general rule which is applicable to many GMP-FS problems in Sect. 4.3.

## 4.1 Specific Window Expansion for CLUSTER EDITING

The following theorems identify certain CLUSTER EDITING instances $G = (V, E)$ as trivial, thereby introducing new invariants for the input.

**Lemma 4.1.** *Given a graph $G = (V, E)$. For an edge $\{u, v\} \in E$, if $u$ and $v$ do not have a common neighbor (i.e., $N(u) \cap N(v) = \varnothing$), one can branch into two cases:*

- *delete $\{u, v\}$, or*
- *delete all edges adjacent to $u$ and $v$, except $\{u, v\}$.*

*Proof.* The completeness of this branching can be seen with an argument very similar to that of Lemma 2.2: Suppose there is a clustering solution $G'$ of $G$ not covered by one of these cases, i.e., the edge $\{u, v\}$ remains, and also at least one other edge adjacent to $u$ or $v$. Consider a clustering solution $G''$ generated from $G'$ by removing all edges adjacent to $u$ and $v$, leaving $\{u, v\}$ as a 2-clique. Now let us compare the cost of editing $G$ to get $G''$ to those of editing $G$ to get $G'$: we need to delete an additional $(\deg_{G'}(u) - 1) + (\deg_{G'}(v) - 1)$ edges; however, $G'$ contains new edges from each neighbor of $u$ (except $v$) to $v$ and vice versa. These edges do not need to be added for $G''$ anymore, saving $(\deg_{G'}(u) - 1) + (\deg_{G'}(v) - 1)$ editing operations.

In summary, editing $G$ to get $G'$ or to get $G''$ incurs the same cost, and $G''$ is covered by Lemma 4.1; therefore, considering other cases is not necessary. □

The following theorem will show that this branching considerably better than the trivial branching from Sect. 2.5, which had a branching number of 2, and therefore is less likely to be the limiting factor for the complexity of a search tree algorithm.

**Theorem 4.1.** *If the branching of Lemma 4.1 is applicable, then we can obtain a branching vector of $(1, 2)$ or better, leading to a branching number of 1.62 or better.*

*Proof.* Assume $\{u, v\} \in E$ and $u$ and $v$ have no common neighbor. If $\deg(u) = \deg(v) = 1$, then $\{u, v\}$ is already a clique and it can be eliminated. Otherwise, at least one of $u$ and $v$ has another neighbor; let us assume w.l.o.g. that $v$ has a neighbor $x$. If $\deg(u) = 1$ and $\deg(v) = 2$, then deleting $\{v, x\}$ is clearly the only reasonable possibility to eliminate the conflict triple $v, u, x$, since the other two possibilities cannot eliminate more than one conflict triple. Therefore, we can assume $\deg(u) + \deg(v) \geq 4$, leading to a $(1, 2)$-branching or better. □

Taking advantage of Theorem 4.1, we can now assume that the input for our Cluster Editing instance is reduced with respect to the applicability of the special branching, and devise a new expansion rule:

**Lemma 4.2.** *Given a window $W = (V_W, E_W)$ of a graph $G$ where $G$ is reduced with respect to Lemma 4.1, and an edge $\{u, v\} \in E_W$, where $u$ and $v$ do not have a common neighbor within $W$. We can then expand $W$ specifically by adding a vertex $x$ which is a common neighbor of $u$ and $v$.*

Ignoring isomorphism, when adding an arbitrary neighbored vertex to a size-$s$ window, there are $2^s - 1$ possible results; since the specific expansion of Lemma 4.2 predetermines two connections to the existing vertices, it produces only $2^{s-2}$ possible results.

This gives rise to a new class of algorithms, which contain specific rules on how to expand the initial $P_3$. We will present details on how to find optimal algorithms within this class in the following section.

## 4.2 Finding Case Distinctions for Cluster Editing

Recall the general search tree algorithm scheme we gave at the beginning of Chap. 3: step (2) requires *expansion rules*, which describe how to expand a window with local information and when to stop expanding and branch according to a branching rule.

In Chap. 3, we only considered one expansion rule: add a vertex neighbored to a vertex of the window, and stop expanding when reaching a certain size. The previous section has shown, though, that other, problem-specific expansion rules are possible.

We base these expansion rules on *expansion steps*. An expansion step tells for a specific window how to expand it with local information.

A expansion rule will first tell which expansion step to apply to the initial window. Depending on the actual input, an expansion step can generate several possible expanded windows. The expansion rule must tell what to do with each possible expanded window: either apply another expansion step, or invoke a branching rule.

With the results from the previous section, we now have several possibilities for an expansion step for Cluster Editing:

- Include an arbitrary vertex which is neighbor of the vertices already in the window.

```
(* The expanders are problem-specific. Example for CLUSTER EDITING. *)
type expander = Expand_new_vertex | Expand_edge of int * int
type algo_step = Branching_rule | Expander of expander * algo list
and algo = { step: algo_step; branching_number: float }
let rec expansion_scheme window =
  if Window.size window > limit
  then { step = Branching_rule;
          branching_number = find_branching_rule window }     (* 1 *)
  else
    let expanders = problem.expanders window in                (* 2 *)
    let algos =
      List.map
        (fun expander →
          let expanded = problem.expand expander window in     (* 3 *)
          let algos = List.map expansion_scheme expanded in    (* 4 *)
          let worst = worst_algo algos in
            { branching_number = worst.branching_number;       (* 5 *)
              step = Expander (expander, algos) })
        expanders
    in
      best_algo algos                                          (* 6 *)
```

Figure 4.2: Pseudo-Code for finding optimal window expansions

- Apply the specific expansion rule of Lemma 4.2, i.e., add the common neighbor of a vertex pair connected with an edge to the window. This is the preferred action, since it leads to a smaller set of expanded windows. Since there might be several vertex pairs in the window matching the criterion, this can provide several possible expansion steps.

From Sect. 3.2, we already know how to find optimal branching rules for step (3) of the general search tree algorithm scheme; we will now focus on finding optimal expansion rules for step (2), which differ in which expansion step is selected for which window.

The basic idea is to take the *best case* of all possible expansions of a window, while assuming the *worst case* of all possible results of an expansion step.

The approach we take in our framework is straightforward. It is based on a recursive procedure *expansion_scheme* (see Fig. 4.2). This procedure takes a window and returns an expansion algorithm represented with the recursive data type *algo*.

An expansion step is represented by the data type *expander*, which is able to represent all possible problem-specific expansion steps for a window.

For Cluster Editing, it can either be *Expand_new_vertex*, meaning the trivial expansion of adding an arbitrary new vertex, or *Expand_edge*, in which case it is parameterized by two integers, denoting the two vertices for which the common neighbor is to be added. The problem-specific function *problem.expand* returns a list of possible expanded windows when given an *expander* and a window.

The recursive data structure *algo* represents a branching rule. It contains an *algo_step*, which describes the action of the branching rule for a certain window, and a *branching_number*, which is required to compare the performance of *algo*s. The first possibility for *algo_step* is *Branching_rule*, meaning the window is not to be expanded any more and the search tree algorithm should branch according to an optimal branching rule. The second possibility is *Expander*, in which case the *algo_step* holds an *expander* and a list of *algo*s, where each element of the list contains the algorithm to be applied for the corresponding element in the list of possible expanded windows returned by *problem.expand*. The *branching_number* is the branching number of a search tree algorithm employing the expansion rule described by the *algo*. The function *best_algo* (*worst_algo*) returns the *algo* with the lowest (highest) branching number from a list.

If the size of the window (for graph modification problems, the number of vertices) exceeds a predefined limit *limit*, an *algo* is returned which simply represents no further expansion and branching according to an optimal branching rule, found by the algorithm from Chap. 3 (1). Otherwise, a list with all possible expanders is collected (2). For Cluster Editing, it will contain one expander which adds an arbitrary vertex, and one expander for each edge where the two vertices it connects do not already have a common neighbor in the window. For each expander, the list of expanded graphs is generated (3). For each expanded graph, an optimal *algo* is calculated recursively (4). The branching number of the resulting *algo* is the worst case of all *algo*s for the possible expanded graphs (5). Finally, from all *algo*s, the best, i.e., the one with the lowest branching number one is returned (6).

Figure 4.3 shows the result of *expansion_scheme* for Cluster Editing when considering graphs up to size 5. For two windows (① and ②), the problem-specific expansion was applied, i.e., the common neighbor of the two endpoints of an edge was added; for window ③, it is not applicable, since for all edges, a common neighbor is already present in the window, and therefore the standard expansion of adding an arbitrary neighbored vertex was applied. The search tree size of a search tree algorithm employing this expansion rule is determined by the worst case of the branching rules found for the windows which are not expanded anymore; in this case, this is window ④, with a branching number of 2.03.

As the figure shows, we will encounter transpositions in the search space; we employed a transposition table in *expansion_scheme* to avoid evaluating a window multiple times.

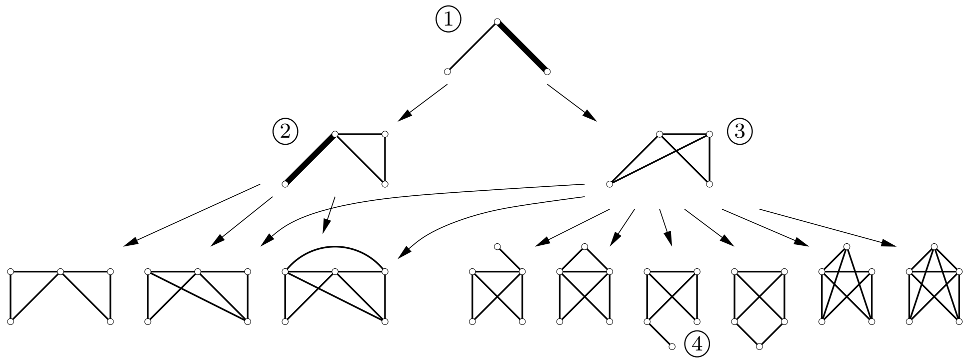

Figure 4.3: Optimal expansion scheme for Cluster Editing with graphs up to size 5. *Bold lines* denote edges for which the specific expansion rule of Lemma 4.2 was applied, i.e., the common neighbor of the endpoints of an edge was added to the window

### 4.2.1 Refined Expansion Thresholds

When looking for expansion schemes, we can speed up the search by utilizing a user-defined cutoff value for the branching number: If a size-$n$ graph $G$ already yields a branching number better than the cutoff value, we omit to try further expansions for $G$. The advantage is two-fold: there are less windows to search when looking for an expansion scheme, and the resulting algorithm has less cases to distinguish. Since the reduced search space might allow to consider expansions up to a larger size within a reasonable time limit, it can even result in improved search tree sizes: for the example of Triangle Vertex Deletion, it allowed us to examine expansions up to 9 vertices instead of 8 vertices, improving the worst-case bound on the size of the resulting search tree from $2.47^k$ to $2.42^k$ (see Table 5.5).

## 4.3 Specific Expansion for Other Graph Modification Problems

As explained in Sect. 3.5, for many GMP-FS problems like Triangle Deletion, a search tree algorithm based on the trivial expansion rule of adding a vertex until the window reaches a certain size does not lead to an algorithm with nontrivial complexity. With an appropriate specific expansion scheme, however, we succeed in finding nontrivial bounds for several of these GMP-FS problems. We require an additional property: An editing operation must not create new conflicts. This is the case for Triangle Editing and all vertex deletion problems, but not, for example, for Cluster Editing or Claw Deletion.

We will use a special branching rule based on the following lemma. We formulate it only for edge editing problems; the extension to other graph

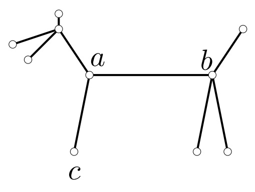

Figure 4.4: Example problem instance (note: not window) for Claw Editing. The edge $\{a, c\}$ is contained in only one induced claw; therefore, deleting it can be omitted from consideration. Analogously, adding $\{b, c\}$ is not useful

modification problems is straightforward.

**Lemma 4.3.** *Given an edge editing problem with characterization by forbidden subgraphs, which has no input instance for which editing any vertex pair creates a new induced forbidden subgraph.*

*In an instance of this problem, if a vertex pair is contained in exactly one induced copy of a forbidden subgraph, it is never useful to edit this vertex pair. If all vertex pairs of the forbidden subgraph can be excluded this way, i.e., it is disjoint from any other copy, no branching is needed, and an arbitrary vertex pair within the forbidden subgraph can be edited.*

*Proof.* Editing any other vertex pair within this conflict graph will also destroy the conflict, and potentially another one. □

Figure 4.4 illustrates this with an example. With this lemma, one can omit one branch in the trivial branching defined in Sect. 3.5 without missing any optimal solutions.

This lemma can be applied for example to Triangle Deletion, since deleting an edge can never create a new triangle; however, it is not applicable, e.g, to Cluster Editing, since adding an edge might create a new $P_3$.

The branching number, when this branching is applicable, is at least one less than the branching number of the trivial branching. For example for Triangle Deletion, we have a branching number of 2 instead of 3.

If we assume the input graph to be reduced with respect to this criterion, we gain a new invariant: every vertex pair of an induced conflict graph must be part of at least one other induced conflict graph. This allows a new specific window expansion rule:

**Lemma 4.4.** *For an edge editing problem with characterization by a forbidden subgraph $F$, given a window $W = (V_W, E_W)$ of a graph $G$ where $G$ is reduced with respect to Lemma 4.1, and a vertex pair $\{u, v\}$ is part of exactly one conflict graph, we can expand the window specifically by adding vertices so that $\{u, v\}$ is part of at least two conflict graphs.*

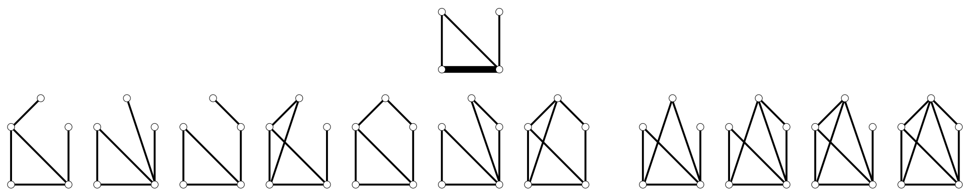

Figure 4.5: Example expansion for Triangle Deletion. The trivial expansion of adding a vertex for the window at the top yields the eleven windows depicted at the bottom. However, Lemma 4.3 tells us that every edge must be part of at least two triangles. When considering this for the edge denoted with a *bold line*, we only get the four windows at the right

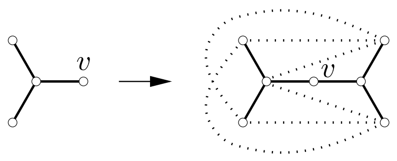

Figure 4.6: Example expansion for Claw Vertex Deletion. Vertex $v$ must be part of at least two claws, because otherwise we would employ a 3-branching on the other three vertices of the claw. Representing this fact in the window requires adding up to three vertices. The *dotted edges* may or may not be present in the input graph, leading to a variety of possible expanded windows

The correctness of Lemma 4.4 is easy to see. Figure 4.5 illustrates the rule for Triangle Deletion.

The branching number of a search tree utilizing this expansion rule is determined by the worse branching number of the following two parts: the branching of Lemma 4.3, and the automatically generated branching with expansion and case distinction. Therefore, with this technique, one cannot get an algorithm with the branching number decreased by more than one compared to the trivial search tree. For example, even if we can come up with a set of branching rules based on the specific expansion which yields a worst-case branching number of 1.7, the branching number of the resulting algorithm is still bound by the 2 from the branching of Lemma 4.3.

When implementing the expansion rule, one has to keep in mind that when looking at a vertex pair in a window which is only part of one conflict graph, it might have only this very vertex pair in common with another conflict graph in the input. Thus, it is required to add several vertices at once in one expansion step to the window to represent the second conflict graph; for a conflict graph of size $s$, up to $s-2$ vertices for editing and deletion problems, and up to $n-1$ vertices for vertex deletion problems (see Fig. 4.6).

# Chapter 5

# Experimental Results

In this chapter, we will present key figures for some algorithms found with our framework for several graph modification problems. For each of these problems, we describe the problem-specific rules implemented within our framework, and present the results of our experiments. Thereby, we measured a variety of values:

**size:** Maximum number of vertices in the considered windows;

**time:** Total running time;

**isom:** Percentage of the running time spent for the isomorphism tests;

**concat:** Percentage of the running time spent for concatenating branching vector sets;

**graphs:** Number of graphs for which a branching rule was calculated;

**maxbn:** Maximum branching number of the computed set of branching rules (determining the worst-case bound of the resulting algorithm);

**avgbn:** Average branching number of the computed set of branching rules; assuming that every induced subgraph appears with the same likelihood, $(\text{avgbn})^k$ would give the average size of the employed search trees, where $k$ is the number of graph modification operations;

**bvlen:** Maximum length of a branching vector occurring in the computed set of branching rules, corresponding to the number of leaves in the branching tree;

**medlen:** Median length of branching vectors occurring in the computed set of branching rules;

**maxlen:** Length of longest branching vector generated in a node of the meta search tree (including intermediary branching vectors);

| | size | time | isom | concat | graphs | maxbn | avgbn | bvlen | medlen | maxlen | bvset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | 4 | <1 sec | 3% | 16% | 5 | 2.42 | 2.33 | 5 | 5 | 8 | 7 |
| (1) | 5 | 2 sec | 2% | 50% | 20 | 2.27 | 2.04 | 16 | 9 | 23 | 114 |
| (1) | 6 | 9 days | 0% | 100% | 111 | 2.16 | 1.86 | 37 | 17 | 81 | 209179 |
| (2) | 4 | <1 sec | 1% | 20% | 6 | 2.27 | 2.27 | 5 | 5 | 8 | 7 |
| (2) | 5 | 3 sec | 0% | 52% | 26 | 2.03 | 1.97 | 16 | 12 | 23 | 114 |
| (2) | 6 | 9 days | 0% | 100% | 137 | 1.92 | 1.80 | 37 | 24 | 81 | 209179 |

Table 5.1: Results for Cluster Editing:
(1) Enumerating all size-$s$ graphs containing a $P_3$;
(2) Expansion scheme utilizing Theorem 4.1

**bvset:** Size of largest branching vector set computed in a node of the meta search tree.

The tests were performed on a 2.26 GHz Pentium 4 PC with 1 GB of main memory running Linux. Memory requirements were up to 300 MB.

## 5.1 Edge Deletion Problems

### 5.1.1 Cluster Editing

Cluster Editing is the *NP*-complete problem that has served as running example in the previous chapters; we summarize the results in Table 5.1. Applying the technique for finding optimal branching trees from Chap. 3, we obtain a worst-case search tree size of $O(2.16^k)$; the application of the problem-specific expansion rule from Sect. 4.1 leads to a worst-case search tree size of $O(1.92^k)$, considerably improving the manually found algorithm of Theorem 2.4. This shows the usefulness of the expansion approach. It underlines the importance of devising a set of good problem-specific rules for the automated approach. Notably, the average branching number avgbn for the computed set of branching rules is significantly lower than the worst-case.

The running time of our program increases as the graph size increases, making it—at the current state of the art—impractical to inspect subgraphs of size larger than six without sacrificing optimality. The typical number of branching cases for a window (medlen) seems high compared to human-made case distinctions (cf. Fig. 2.4 and 2.5, with 5 cases each), but should pose no problem for an implementation.

The main reason for the high running times is that the case distinction becomes more and more complicated as the sizes of considered graphs increase. As one consequence of the more complicated case distinction, the program has to do much more branching vector concatenations (refer to the drastic increase of value bvset in Table 5.1). As we have stated in Sect. 3.3, besides the subsumption mechanism, we have not found an efficient way to determine the best concatenation of two sets of branching vectors other than basically trying all possibilities. It can be observed from Table 5.1 that, for graphs with six vertices, the program spends almost all its running time on the concatenations of branching vectors; branching vector sets can contain huge amounts of incomparable branching vectors (bvset), and a single branching vector can get comparatively long (maxlen). Hence, one of the most challenging tasks to improve our program is a solution of this problem. This is also the reason that graph isomorphism testing, perhaps surprisingly, contributes a decreasing proportion (isom) to the running time when increasing the graph size.

Summarizing the results together with the findings about a problem kernel [GGHN03b], we have the following theorem:

**Theorem 5.1.** CLUSTER EDITING *can be solved in* $O(1.92^k + |V|^3)$ *time.* □

Observe that CLUSTER EDITING is equivalent to unweighted CORRELATION CLUSTERING on complete graphs as studied in [BBC02]. The best known polynomial-time approximation algorithm for minimizing the number of of edge modifications yields an approximation factor of only 4 [CGW03], giving particular importance to exact fixed-parameter algorithms for this problem.

### 5.1.2 Cluster Deletion

CLUSTER DELETION is the special case of CLUSTER EDITING where only edge deletions are allowed. It was shown to be *NP*-complete by Natanzon [Nat99]. In a previous work [GGHN03b], an algorithm for CLUSTER DELETION with $O(1.77^k)$ search tree size was given.

Because of the close relation to CLUSTER EDITING, the problem-specific expansion scheme of Sect. 4.1 is also applicable to CLUSTER DELETION. However, it turns out that the search tree size of the optimal search trees is low enough that the time complexity of the resulting algorithm is determined by the $(1, 2)$-branching of Theorem 4.1. Therefore, to improve the overall time complexity, it would be desirable to improve upon this branching with a more careful case distinction. We will now present such an improvement.

**Lemma 5.1.** *Given a graph* $G = (V, E)$*. If there is an edge* $\{u, v\} \in E$ *where* $u$ *and* $v$ *have no common neighbor, then we can in* CLUSTER DELETION *apply a branching rule with the branching vector* $(1, 3)$.

*Proof.* As shown in the proof for Theorem 4.1, there is no need for branching if $\deg u + \deg v \leq 3$. If $\deg u + \deg v \geq 5$, we have at least a $(1, 3)$-branching by Lemma 4.1: delete either $\{u, v\}$ or all other edges adjacent to $u$ and $v$. It remains the case of $\deg u + \deg v = 4$. We distinguish two subcases:

| | size | time | isom | concat | graphs | maxbn | avgbn | bvlen | medlen | maxlen | bvset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | 4 | <1 sec | 12% | 12% | 5 | 1.77 | 1.65 | 4 | 2 | 5 | 4 |
| (1) | 5 | <1 sec | 37% | 22% | 20 | 1.63 | 1.52 | 8 | 2 | 13 | 83 |
| (1) | 6 | 6 min | 4% | 92% | 111 | 1.62 | 1.43 | 16 | 2 | 35 | 7561 |
| (2) | 4 | <1 sec | 7% | 15% | 6 | 1.77 | 1.70 | 4 | 2 | 5 | 4 |
| (2) | 5 | <1 sec | 11% | 33% | 26 | 1.63 | 1.54 | 8 | 2 | 13 | 83 |
| (2) | 6 | 6 min | 0% | 97% | 137 | 1.53 | 1.43 | 16 | 2 | 35 | 7561 |

Table 5.2: Results for Cluster Deletion:
(1) Enumerating all size-$s$ graphs containing a $P_3$;
(2) Expansion scheme utilizing Lemma 5.1

(1) Vertex $u$ has two neighbors $x, y \in N(u)$ with $x \neq y$, $x \neq v$ and $y \neq v$.

(1.1) If $\{x, y\} \notin E$, then we can delete $\{u, x\}$ and $\{u, y\}$, since at least two of the edges $\{u, v\}$, $\{u, x\}$, and $\{u, y\}$ have to be deleted and deletion of $\{u, x\}$ and $\{u, y\}$ does not affect any solution of the problem instance.

(1.2) If $\{x, y\} \in E$, then we assume that there is at least one vertex $z \in V$ and $z \neq u$ which is a neighbor of $x$ or $y$, since otherwise, $u, v, x, y$ form an isolated component. It is easy to observe that deleting only one of the edges $\{u, x\}$ and $\{u, y\}$ can never be better than deleting both of them or keeping them both and deleting $\{u, v\}$ and the edges adjacent to $x$ and $y$ but different from $\{x, y\}$, $\{u, x\}$, and $\{u, y\}$. Since there is at least one edge between $y$ and $z$ or $x$ and $z$, we get a branching of at least $(2, 2)$, which is better than a $(1, 3)$-branching.

(2) Each of the vertices $u$ and $v$ has an additional neighbor, i.e., there are vertices $x \in N(u)$ and $y \in N(v)$ with $x \neq y$, $x \neq v$ and $y \neq u$.

(2.1) If one of $x$ and $y$ has no neighbor besides $u$ and $v$, then we delete $\{u, v\}$. This is justified as follows: Assume w.l.o.g. that $y$ has no neighbor besides $v$. For the case that $x$ and $u$ are in the same clique of the clustering solution, we have to delete $\{u, v\}$; for the case that they are not, we have to delete either $\{u, v\}$ or $\{v, y\}$. Therefore, deleting $\{u, v\}$ is always a correct solution for the subgraph consisting of $u$, $v$, and $y$.

(2.2) If vertex $x$ has more than one neighbor, then we consider the edge $\{u, x\}$. Since $u$ and $x$ have no common neighbor and $\deg u + \deg x \geq 5$, we can achieve at least $(1, 3)$-branching based on $\{u, x\}$.

(2.3) If both $x$ and $y$ have exactly one neighbor, both different from $u$ and $v$, then we make a branching into two cases. The first case is

| | size | time | isom | concat | graphs | maxbn | avgbn | bvlen | medlen | maxlen | bvset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | 4 | < 1 sec | 2% | 19% | 4 | 2.57 | 2.47 | 6 | 5 | 10 | 8 |
| (1) | 5 | 6 sec | 0% | 83% | 19 | 2.47 | 2.34 | 14 | 5 | 45 | 530 |

Table 5.3: Results for TRIANGLE DELETION:
(1) Expansion scheme utilizing specific expansion rule

that $v$ and $y$ are in the same clique of the clustering solution. For this case, we have to delete the edge adjacent to $v$ and $y$ different from $\{v, y\}$. For the second case that $v$ and $y$ are not in the same clique, we have to delete $\{v, y\}$. The resulting subgraph, which consists of $u$, $v$, $x$ and the other neighbor of $x$, satisfies the assumption of case (2.1) for the edge $\{u, x\}$. Therefore, we can delete $\{u, x\}$. Summarizing the two cases, we have a $(2, 2)$-branching.

In summary, we have a (1, 3)-branching in the worst case. □

### 5.1.3 TRIANGLE DELETION

The TRIANGLE DELETION is the edge deletion problem with the triangle ($C_3$) as forbidden subgraph. Since the forbidden subgraph is a clique, the completion and editing problems are not meaningful.

**Problem-specific rules.** We follow the general scheme from Sect. 3.5. To get a nontrivial search tree (i.e., one with better than $O(3^k)$ size), the specific expansion from Sect. 4.3 is required.

**Discussion.** This problem is an example where the mechanized analysis so far could not improve an existing search tree algorithm. Since TRIANGLE DELETION can be reduced to 3-HITTING SET, we can solve it using an existing search tree algorithm for 3-HITTING SET having a worst-case branching number of 2.27 [NR03a], whereas the worst-case branching number determined by our analysis is 2.47 when considering windows up to size five. We are confident, however, that, by additional reduction rules or the use of heuristics for branching vector concatenation, beating the 2.27 bound is feasible.

## 5.2 Vertex Deletion Problems

We have also applied our automated approach to several vertex deletion problems that have a forbidden set characterization, with the set containing a single forbidden subgraph. As shown by Lewis and Yannakakis [LY80], all these problems are *NP*-complete.

Vertex deletion problems defined by a single forbidden induced subgraph of size $d > 1$ can be reduced to the *NP*-complete $d$-Hitting Set problem, which is defined as follows:

**Problem 5.1.** ***Input:*** *A collection $C$ of subsets of size $d$ of a finite set $S$ and a positive integer $k$.*

***Question:*** *Is there a subset $S' \subseteq S$ with $|S'| \leq k$ such that $S'$ contains at least one element from each subset in $C$?*

Note that $d$-Hitting Set can be seen as a generalization of Vertex Cover, which is equivalent to 2-Hitting Set.

**Lemma 5.2.** *For a* Vertex Deletion *problem where the graph property is defined by a forbidden induced subgraph of size $d$, one can find an $O(n^d)$ time many-one reduction to $d$-*Hitting Set.

*Proof.* For a given instance of the Vertex Deletion problems consisting of a graph $G = (V, E)$ and an integer $k$, we construct a finite set $S$ such that each element in $S$ corresponds to a vertex in $V$, i.e., $n := |S| = |V|$. For a forbidden subgraph with $d$ vertices, we can trivially enumerate all occurrences of the forbidden subgraph as induced subgraph and, for each of the occurrences construct a set consisting of $d$ elements; each of the elements corresponds to a vertex in the occurrence. These $d$-element sets form the collection $C$. Then, we have an instance of the $d$-Hitting Set problem. It is easy to observe that the Vertex Deletion instance can be solved with at most $k$ vertex deletions iff the constructed instance of $d$-Hitting Set has a solution $S'$ with $|S'| \leq k$. The main part of the reduction is to enumerate all occurrences of the forbidden size-$d$ subgraph and, hence, can be done in $O(n^d)$ time. □

We considered the two problems Cluster Vertex Deletion and Triangle Vertex Deletion, which are vertex deletion problems with the size-3 graphs $P_3$ and $C_3$ as forbidden subgraph, respectively. As outlined above, these problems, therefore, can be reduced to the *NP*-complete 3-Hitting Set. For 3-Hitting Set, an elaborate search tree algorithm is known with $O(2.27^k + |C|)$ running time [NR03a]. We utilized the reduction rules for GMP-FS problems from Sect. 3.5. As expansion rule, the generic method of adding an arbitrary vertex to the window was applied.

We also examined two vertex deletion problems with forbidden subgraphs of size 4, namely Claw Vertex Deletion, with a claw as forbidden subgraph, and Cograph Vertex Deletion, with the $P_4$ as forbidden subgraph. Cograph modification problems in particular seem like an interesting problem because, when restricted to cographs, many *NP*-hard problems are solvable in polynomial time [CPS85].

| | size | time | isom | concat | graphs | maxbn | avgbn | bvlen | medlen | maxlen | bvset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | 4 | $<$ 1 sec | 3% | 16% | 5 | 2.42 | 2.37 | 4 | 3 | 6 | 3 |
| (1) | 5 | $<$ 1 sec | 6% | 14% | 20 | 2.31 | 2.16 | 4 | 4 | 10 | 7 |
| (1) | 6 | 1 sec | 8% | 12% | 111 | 2.31 | 1.98 | 6 | 4 | 14 | 24 |
| (1) | 7 | 26 sec | 19% | 14% | 852 | 2.27 | 1.86 | 6 | 4 | 21 | 65 |
| (1) | 8 | 39 min | 34% | 12% | 11116 | 2.27 | 1.76 | 10 | 5 | 32 | 289 |
| (2) | 4 | $<$ 1 sec | 12% | 3% | 6 | 2.42 | 2.37 | 4 | 3 | 6 | 3 |
| (2) | 5 | $<$ 1 sec | 3% | 15% | 26 | 2.31 | 2.16 | 4 | 4 | 10 | 7 |
| (2) | 6 | $<$ 1 sec | 0% | 22% | 74 | 2.31 | 2.06 | 6 | 4 | 13 | 12 |
| (2) | 7 | $<$ 1 sec | 0% | 27% | 119 | 2.27 | 2.02 | 6 | 4 | 19 | 49 |
| (2) | 8 | 5 sec | 0% | 38% | 205 | 2.27 | 2.00 | 8 | 4 | 25 | 146 |
| (2) | 9 | 46 sec | 0% | 53% | 367 | 2.26 | 1.92 | 9 | 4 | 37 | 534 |
| (2) | 10 | 7 min | 0% | 69% | 681 | 2.26 | 1.90 | 11 | 4 | 48 | 2422 |

Table 5.4: Results for CLUSTER VERTEX DELETION:
(1) Enumerating all size-$s$ graphs containing a $P_3$;
(2) Expansion scheme with cutoff (see Sect. 4.3)

| | size | time | isom | concat | graphs | maxbn | avgbn | bvlen | medlen | maxlen | bvset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | 5 | $<$ 1 sec | 2% | 17% | 9 | 2.57 | 2.24 | 5 | 4 | 11 | 10 |
| (1) | 6 | $<$ 1 sec | 2% | 25% | 44 | 2.57 | 2.24 | 7 | 4 | 19 | 37 |
| (1) | 7 | 7 sec | 0% | 32% | 447 | 2.47 | 2.10 | 10 | 4 | 30 | 121 |
| (1) | 8 | 9 min | 0% | 46% | 7225 | 2.47 | 1.97 | 13 | 5 | 42 | 384 |
| (2) | 8 | 23 sec | 0% | 43% | 433 | 2.47 | 2.10 | 13 | 4 | 34 | 355 |
| (2) | 9 | 10 hours | 0% | 56% | 132370 | 2.42 | 1.97 | 17 | 5 | 66 | 1842 |

Table 5.5: Results for TRIANGLE VERTEX DELETION:
(1) Expansion scheme utilizing problem-specific expansion rule;
(2) additionally, with cutoff

**Discussion.** We show detailed results for Cluster Vertex Deletion in Table 5.4 and for Triangle Vertex Deletion in Table 5.5.

Using the enumeration without nontrivial expansion for Cluster Vertex Deletion, we could only process graphs with up to eight vertices since the number of graphs to be inspected is huge. This yields the same worst-case branching number 2.27 as we have from the 3-Hitting Set algorithm [NR03a]. Using a cutoff value (see Sect. 4.2.1) reduces the number of graphs to be inspected drastically and, thus, allows us to inspect graphs with up to ten vertices. In this way, we can improve the worst-case branching number slightly to 2.26.

When comparing the two approaches, we observe that, when using cutoff values, the average branching number (avgbn) of the computed set of branching rules becomes larger compared to the case where cutoff values were not used. The explanation is that the branching is not further improved as soon as it yields a branching number better than the cutoff value. When implementing the computed search tree algorithm, however, a better average branching number might be more desirable than a better worst-case branching number.

For both Claw Vertex Deletion and Cograph Vertex Deletion, we examined windows up to size 7; the best algorithm found for Claw Vertex Deletion has a branching number of 3.55, and the best for Cograph Vertex Deletion has a branching number of 3.30. This means we could not beat the branching number of 3.30 provided by the algorithm for 4-Hitting Set; however, at least for Cograph Vertex Deletion, we can match its performance with a lot less effort.

## 5.3 Bounded Degree Dominating Set

Domination in graphs is among the most important problems in combinatorial optimization [HHS98a, HHS98b]. The underlying *NP*-complete decision problem Dominating Set is defined as follows:

**Problem 5.2.** ***Input:*** *A graph $G = (V, E)$ and a positive integer $k$.*

***Question:*** *Does $G$ have a* dominating set *of size at most $k$, that is, a subset $V' \subseteq V$ of vertices such that every vertex in $V - V'$ is adjacent to some vertex in $V'$?*

Dominating Set remains *NP*-complete even when considering only special graph classes, like planar graphs or graphs in which the degree of $G$ is bounded by a constant $B \geq 3$. Here, we consider the case of graphs with degree bounded by 3. This restriction allows to give a size bound for a trivial search tree algorithm: for any non-dominated vertex $v$, either take $v$ into the dominating set, or one of its neighbors. This search tree has size $O(4^k)$.

For our framework, we first introduce a vertex annotation “white”, which marks vertices that are already dominated, but might still be chosen to be in the dominating set to dominate other vertices. The branching objects are black (i.e., non-white) vertices: As sub-branching, take either a vertex $v$ into the dominating set, or, for each neighbor of $v$, take this neighbor. When taking a vertex, we remove it from the window and mark all neighbors white. To be able to properly execute this sub-branching, all neighbors of $v$ need to be represented in the window; therefore, we introduce an annotation which tracks the exact degree of each node in the window.

These ingredients would be already enough to reproduce the trivial search tree algorithm. For additional gain, we need reduction rules. Alber et al. [AFF$^+$01] give several simple reduction rules in the context of a problem kernel reduction, for example:

- Delete edges between white vertices.
- Delete a degree-1 white vertex.
- If there is a degree-1 black vertex $w$ with neighbor $u$ (either black or white), then delete $w$ and place $u$ in the dominating set.

We implemented all of the applicable reduction rules of [AFF$^+$01].

The expansion rule adds the neighbor of a vertex which does not yet have all of its neighbors represented in the window (as determined with the exact degree annotation). One has to consider all combinations of whiteness, degree and connections to other vertices of this new vertex to other vertices in the window.

When expanding up to graphs of size 5, we could determine a search tree algorithm with search tree size $O(3.79^k)$, where $k$ denotes the number of vertices in the dominating set. When considering larger sizes, the computation time to find an optimal branching for a specific window became prohibitive; this can be explained with the higher number of cases in a sub-branching as compared to graph modification problems. Conceivably, heuristics for finding non-optimal branching rules to allow examination of larger expanded windows would make sense here; initial experiments with a very simple heuristics indicate a search tree size of $O(3.71^k)$.

## 5.4 Summary

Focusing on the worst-case branching numbers computed for various graph modification problems, we give an overview on our results in Table 5.6: We compare the worst-case branching numbers corresponding to a trivial branching, the best so far known result, and the search tree algorithm computed by our method.

In Fig. 5.1, we compare, for different graph modification problems, the decrease of the worst-case branching numbers when increasing the size of the considered subgraphs. In most cases, inspecting larger subgraphs yields

| Problem | Trivial | Known result | New |
|---|---|---|---|
| CLUSTER EDITING | 3 | 2.27 [GGHN03b] | 1.92 |
| CLUSTER DELETION | 2 | 1.77 [GGHN03b] | 1.53 |
| CLUSTER VERTEX DELETION | 3 | 2.27 [NR03a] | 2.26 |
| TRIANGLE DELETION | 3 | 2.27 [NR03a] | 2.47 |
| TRIANGLE VERTEX DELETION | 3 | 2.27 [NR03a] | 2.42 |
| COGRAPH VERTEX DELETION | 4 | 3.30 [NR03a] | 3.30 |
| BOUNDED DEGREE DOMINATING SET | 4 | | 3.71 |

Table 5.6: Summary of search tree sizes for the problems considered

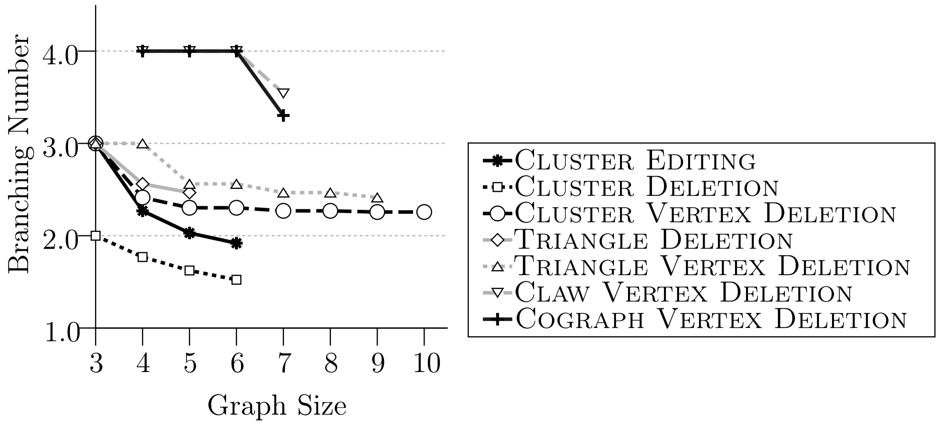

Figure 5.1: Worst-case branching number depending on size of considered subgraphs

an improved worst-case branching number. There are exceptions, however: For example, in TRIANGLE VERTEX DELETION, we observe that we do not improve the worst-case branching number when going from graph size 3 to 4, from 5 to 6, and from 7 to 8, but only from 4 to 5, 6 to 7, and 8 to 9. This is caused by the problem-specific expansion rule 4.3, which needs to add two vertices at once.

## 5.5 Outlook

In this section, we will give details for possible applications of our framework to several further problems.

### 5.5.1 VERTEX COVER

The VERTEX COVER problem is one of the six "basic" *NP*-complete problems according to Garey and Johnson [GJ79]. It is certainly among the best-studied combinatorial problems.

**Problem 5.3.** ***Input:*** *A graph $G = (V, E)$.*

***Question:*** *A vertex cover of $G$, i.e., a subset $V' \subseteq V$ such that, for each edge $\{u, v\} \in E$ , at least one of $u$ and $v$ belongs to $V'$.*

For us, Vertex Cover is of special interest because of its simple structure, and because very refined search tree algorithms are known, the best of which has $O(1.286^k + kn)$ running time [CKJ01, NR99, NR03b], where $k$ is the number of vertices in the vertex cover, and $n$ is the total number of vertices in the input.

A sensible representation of windows is clearly an induced subgraph. As annotations, the exact degree of each vertex seems useful; for vertices with high ($\geq$ 6) or low ($\leq$ 2) degree, good branchings are known. The subbranching consists of two cases: either take a vertex $v$ into the vertex cover (and remove it from the window), or take all of its neighbors (and remove them).

Many reduction rules for Vertex Cover have been devised, for example "vertex folding" [CKJ01], which gets rid of certain degree-2 vertices. These reduction rules usually can be applied when given only the window view on the instance, making Vertex Cover a hopeful candidate for employment of our framework.

### 5.5.2 X3SAT

Because of their manifold applications, satisfiability problems have been subject of intense research in the past [Sch01, DHIV01]. The Exact 3-Satisfiability problem is defined as follows:

**Problem 5.4 (X3SAT).** ***Input:*** *A set $V$ of $n$ variables and a collection $C$ of conjunctive normal form clauses over $V$, i.e., $C$ is a disjunction of clauses $c_i : C = c_1 \vee c_2 \vee \cdots \vee c_m, m \in \mathbb{N}$, and each clause $c_i$ is a conjunction of at most 3 literals over $V$.*

***Question:*** *Is there a truth assignment $t : V \to \{0, 1\}$, such that exactly one literal in each clause is set to true?*

X3SAT is *NP*-complete. Hirsch and Kulikov [HK02] have given an algorithm which solves it in $O(1.1194^n)$ time. Their algorithm also follows the well-known search tree scheme of reduction and branching.

As window, we propose a subset of the clauses. An annotation for each variable, which captures the number of positive and negative occurrences of a variable, or at least bounds of this variable, seems advisable. The subbranching is obvious: For a variable $v$, branch into two cases with $v$ set to true and $v$ set to false. Regarding reduction rules, one can find a large collection in the paper by Hirsch and Kulikov.

Adapting our framework to X3SAT or similar satisfiability problems seems interesting, because it would demonstrate applicability beyond graph problems. Also, comparisons with recent automated case distinction approaches for satisfiability problems [NS03, FK03] are desirable.

# Chapter 6

# Implementation

The source of our implementation is divided into two roughly equal sized parts. The first part, written in about 1500 lines of low-level C, handles the graph representation and the interface to the **nauty** library [McK90], which provides generation of canonical representations of graphs for isomorphism tests and hash table operations. For speed reasons, it also provides some problem-specific operations like checking whether a graph contains a $P_3$.

The second part, which implements the advanced algorithms like the meta search tree, the graph enumeration, and the expansion scheme is written in Objective Caml [LVD$^+$96], a high-level functional programming language. Features like automatic memory management, closures, and powerful data structure representation allowed for much faster and easier development than an early C++ prototype did.

Glue code provides the functionality of the C code as an abstract `graph` data type. All operations on `graph`s are functional-style, i.e., `graph`s are never modified, but each operation creates a new graph. This might result in some performance loss, because it puts more load on the garbage collector, but was certainly helpful in avoiding bugs and resulted in cleaner code.

## 6.1 Graph Representation

A graph with $n$ vertices is represented as $n$ words, where bit $i$ in word $j$ is set iff the edge $\{i, j\}$ exists. Clearly this representation is redundant, since, allowing only undirected graphs, bit $i$ in word $j$ and bit $j$ in word $i$ must always have the same value; also, disallowing self-loops, bit $i$ in word $i$ must always be 0. However, this representation allows very fast tests for the existence of an edge, fast iteration over all neighbors, and also simplifies some more complicated algorithms like the test for connectedness. Also, it can be directly used by **nauty**.[1]

[1]Note that for compatibility with **nauty**, the bits are numbered starting with the most significant bit.

A limitation of this representation is that the number of vertices is limited by the maximum size of a word, which is 64 for the C language. Since our algorithms usually enumerate at least a noticeable percentage of all graphs up to size $n$, this does not seem to be a serious limitation (all graphs we studied had less than 20 vertices). The limitation could also be removed with moderate effort by using multiple words per vertex.

Our graph representation allows to attach an arbitrary amount of attribute bits to both vertices and vertex pairs. For example, CLUSTER EDITING requires an "immutable" bit for each vertex pair. Vertex pair attribute bits are represented like graphs as vectors of $n$ words; in fact, the edges of a graph are simply represented by a special vertex pair attribute bit. Vertex attributes are stored with as a single word per attribute, with one bit per vertex. For example, BOUNDED DEGREE-3 DOMINATING SET requires three bits: one to store the color ("black" or "white") and two to store the degree of the vertex.

The code is organized so that graph modification problems with forbidden subgraph characterizations can be added fairly easy; it would even be feasible to modify it to accept arbitrary forbidden subgraphs and generate the necessary functions automatically.

## 6.2 Representing Branching Vector Sets

As explained in Sect. 3.2, the following function *bvset_merge* can take a large percentage of the total running time:

*bvset_merge*: Given three sets of branching trees $B$, $N_1$, and $N_2$.[2] Let $N_1 \times N_2$ denote the set of all possible concatenations of a branching vector from $N_1$ with a branching vector from $N_2$. Calculate $B' = \text{prune}(B \cup (N_1 \times N_2))$, where $\text{prune}(S) = \{v \in S : \nexists x \in S : x \text{ subsumes } v\}$ (recall Sect. 3.3 for the definition of "subsume").

This operation has to be carried out in each node of the meta search tree, for each branching object; $B$ is a set of possible branching rules gained from considering other branching objects, and $N_1$ and $N_2$ are sets of branching rules for the current window when branching according to the current branching object into the first and second branch.

While, from an algorithmic viewpoint, we are dealing with sets of branching rules represented as branching trees, for the *bvset_merge* operation, only their branching vectors are relevant. If one is only interested in the branching number of an optimal branching rule, one could even discard the branching trees and only pass their branching vectors. Therefore, we will only talk about "branching vector sets" in the following.

[2]We will again, for simplicity, only depict the case of branching rules based on subbranchings into exactly two cases.

(1, 1, 4)
(1, 2, 3)
(1, 3, 3, 4)
(2, 2, 2, 3)
(2, 2, 3, 3, 4)
(2, 3, 3, 3, 4, 4)
(3, 3, 3, 3, 4, 4, 4)

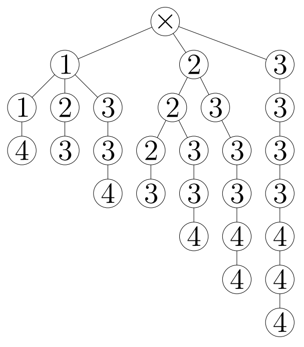

Figure 6.1: Trie representation of a branching vector set. This example represents the set of branching vectors the meta search tree finds for Cluster Editing with a $P_5$ as window

A simplistic implementation would represent a branching vector set as an array of branching vectors. An obvious improvement is to keep these vectors sorted at all time, since that simplifies comparing them.

The operation *bvset_merge* can then be implemented as follows: Initialize $B'$ with $B$. For each possible concatenation $v$ of two vectors from $N_1$ and $N_2$, first check if it is subsumed by any vector from $B'$. If so, discard it. Otherwise, discard all vectors from $B'$ which are subsumed by $v$, and add $v$ to $B'$. (In practice, these two operations can be done in parallel.)

Consider sets of $n$ branching vectors, each of length $l$ (in our measurements, we encountered values for $n$ up to 200,000 and for $l$ up to 80). With the simple array representation, we have:

- $O(n \cdot l)$ memory usage;
- $O(l)$ time for comparing two branching vectors;
- $O(n^3 \cdot l)$ time for *bvset_merge* (generate each concatenation, and compare against each existing branching vector).

In particular because of the very inefficient merging (which has to be done in each meta search tree node), better data structures are needed. One possibility is to use a *trie*, which is a tree where each node is labeled with an integer, and each path from the root to a leaf represents a branching vector (see Fig. 6.1). Since a branching vector cannot be a prefix of another incomparable one, no marking of vector ends in internal node is needed.

This has several advantages: Firstly, common prefixes are represented in a compact form, reducing memory requirements. Secondly, the required operations become much faster: We still generate each concatenation $v$ of two vectors. We can then, in parallel, check whether $v$ is subsumed by any vector represented in $B$, or whether any vector in $B$ is subsumed by $v$ and must be discarded. For this, we traverse both the vector and the trie in parallel, very similar to the comparison of just two branching vectors.Each time we get an "incomparable" result, there is no need to further traverse the trie, therefore skipping further comparison with a potentially large amount of branching vectors.

Because of the much increased overhead compared to the simple array storage, the trie representation only amortizes when working with very large branching vector sets (several 1000 vectors). Therefore, for simple problems, where the meta search tree is shallow, it might be more efficient to employ the array representation, or a hybrid scheme that switches to tries once the sets reach a certain size.

An early prototype in C++ used the array representation; the current Ocaml code only implements the trie representation, for reasons of simplicity. Evaluating the trade-offs is subject of future work. Also, it seems feasible to speed up *bvset_merge* further by not generating all pairs of concatenations explicitly, but to traverse $N_1$ and $N_2$ intelligently.

## 6.3 Invocations

The main program, which is called `ce-branch`, can be called in three different basic modes:

- Find an optimal branching tree for a particular window (default);
- Find branching trees for all windows of a certain size (`-e` option);
- Expansion scheme up to a certain size (`-x` option).

The problem is selected with the `-p` option. Currently implemented are:

- `ce`: Cluster Editing
- `cd`: Cluster Deletion
- `cvd`: Cluster Vertex Deletion
- `td`: Triangle Deletion
- `tvd`: Triangle Vertex Deletion
- `clawvd`: Claw Vertex Deletion
- `p4vd`: Cograph Vertex Deletion
- `squarevd`: Square Vertex Deletion
- `bd3ds`: Bounded Degree-3 Dominating Set

We will give a few examples for the modes. Single graphs can be input in two ways: on standard input, where each input line consists of two integers denoting two vertices connected with an edge, or as "GraphID". A GraphID is an integer which compactly encodes undirected graphs with a binary representation, where bit 0 is set if the edge $\{0, 1\}$ is present, bit 1 for $\{0, 2\}$, bit 3 for $\{1, 2\}$, and so on; generally, the edge $\{u, v\}, u < v$ is encoded in bit $\sum_{i=0}^{v-1} i + u$. A GraphID is passed with the `-n` option and its decimal representation:

```
% ./ce-branch -p ce -n 880
{
0 4 connected
1 3 connected
2 3 connected
2 4 connected
3 4 connected
} cost = 0
ANSWER:
[[1 1 1] 3.000000
 [1 1 2 3] 2.546818
 [1 1 2 5 5] 2.460890
...
 [2 3 3 3 3 3 3 3 3 4 5] 2.215213
 [2 3 3 3 3 3 3 3 4 4 5 5] 2.191183
]
Best branching vector:
[1 2 3 3 4 5 5] 2.120334
({0, 4}: ({0, 2}: ({0, 3}: ({0, 1}: *, *), *),
                  ({0, 3}: ({0, 1}: *, *), *)), *)
```

The first part of the output, between curly braces, is the input graph (in this case, the one corresponding to the GraphID 880). The next part lists all incomparable branching vectors computed by the root of the meta search tree. Then, the best one is displayed, along with a dense representation of the corresponding branching tree in the format `(`*branch_edge*`:` *left_branch*`,` *right_branch*`)`, where a `*` marks a leaf. Edges are in the format `{`*u*`,` *v*`}`, where *u* and *v* are integers denoting vertices.

This branching tree is also depicted in Fig. 6.2; the vertices are numbered clockwise with 0 at the top.

With the `-f` option, a figure of the branching tree in `xfig` format is printed to the standard output. For example, the following invocation yields the picture in Fig. 6.2:

```
% ./ce-branch -n 880 -f > test.fig
```

When invoked with the `-e` or `-x` option, the program prints one line for every window for which a branching rule is calculated (lines wrapped here for reading convenience):

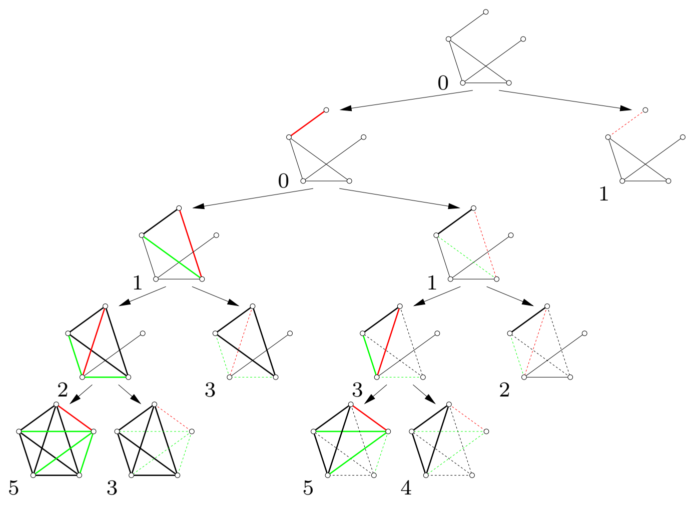

Figure 6.2: Example output of out branching tree finder. For each window, the branching object for which a branch was just performed is marked *red*; *green* color is used to mark vertex pairs which were changed due to application of branching rules

```
% ./ce-branch -e 4
   30 2.26953084 0 6 8 336 5
      ({0, 2}: ({0, 1}: ({0, 3}: *, *), ({0, 3}: *, *)), *)
   44 2.41421356 0 7 8 336 5
      ({2, 3}: ({0, 2}: ({0, 1}: *, *), ({1, 2}: *, *)), *)
   56 2.41421356 0 5 8 336 5 ({0, 3}:
      ({0, 1}: ({0, 2}: *, *), ({0, 2}: *, *)), *)
   60 2.26953084 0 5 6 336 5
      ({0, 3}: ({0, 1}: ({0, 2}: *, *), ({0, 2}: *, *)), *)
   62 2.26953084 0 5 5 336 5
      ({0, 1}: *, ({0, 2}: ({0, 3}: *, *), ({0, 3}: *, *)))
```

We will explain the meaning of the columns with the first line as example:

- `30`: GraphID of the window;
- `2.26953084`: branching number;
- `0`: run time in seconds for this window (rounded to next integer);
- `6`: maximum size of a branching vector set;
- `8`: maximum length of a branching vector;
- `336`: number of branching vector set merge operations;
- `5`: length of the optimal branching vector;
- rest: compact representation of an optimal branching tree.

## 6.4 Adding Problems

The process of adding a new problem is not as smooth yet as it could be. In particular, it is assumed that the window is an annotated graph, and problems with sub-branching into more than two cases require some source changes and recompilation.

The main tasks when adding a new problem are:

1. Deciding on annotations.
2. Identifying the initial window.
3. Identifying branching objects.
4. Writing the sub-branching function.

We will illustrate the process by an example: the Square Vertex Deletion problem, i.e., the vertex deletion problem with the square ($C_4$) as forbidden induced subgraph.

The main task is to create an instance of the *Problem.t* structure defined in `problem.ml`. Some of the more important fields are:

***vertex_ bits/edge_ bits*:** number of bits for annotation of vertices and edges, respectively. Here, one bit is needed for the "immutable" vertex annotation; as a little quirk, the implementation stores the edges of the graph as vertex pair annotations, and therefore also one vertex pair bit (slightly misleading called *edge_ bits*) is needed.

***conflict_ graph*:** A function which returns an initial window. Here, we return a graph containing a square, with no annotations.

***fold_ branch_ objs*:** A function which, given a window, iterates over all its valid branching objects. Here, we want to branch on all mutable vertices with degree at least one (in the implementation, degree 0 vertices are not considered to be part of a window, but rather dummies).

***branch_ 1/branch_ 2*:** The two cases of a sub-branching.[3] Here, for the first sub-branching case (*branch_ 1*), the generic function *del_ vertex* can be used, which deletes a vertex and all adjacent edges. The second sub-branching case (*branch_ 2*) requires a problem-specific function, since after marking a vertex permanent, the reduction rules from Sect. 3.5 need to be applied.

***expanders*:** A list of expanders for the expansion scheme. Here, we can either use the problem-specific expander based on results from Sect. 4.3, or the generic expander which adds a vertex to the window.

[3]Other numbers of cases in sub-branchings require slight code modifications.

```
let square_vertex_deletion = {
  name = "square-vertex-deletion";
  vertex_bits = 1;
  edge_bits = 1;
  conflict_graph = fun size vb eb →
    List.fold_left
      (fun g (i, j) → Graph.set_connected g i j)
      (Graph.make size vb eb)
      [(0, 1); (1, 2); (2, 3); (3, 0)];
  fold_branch_objs =
    vertex_fold (fun g i →
      not (Graph.is_immutable_vertex g i)
      && not (Graph.is_deg_0 g i));
  branch_1 = fun g (Branch.Vertex i) → Graph.del_vertex g i;
  branch_2 = fun g (Branch.Vertex i) → Graph.svd_permanentize g i;
  expanders = [ Algo.Vertex_expander square_expander;
                Algo.Expander add_vertex_expander ]; }
```

Figure 6.3: Problem definition for Square Vertex Deletion

The actual code for Square Vertex Deletion is given in Fig. 6.3.

In addition to this structure, the option parsing in `main.ml` has to be adapted.

# Chapter 7

# Conclusion

We presented a software tool to automatically generate exact search tree algorithms for graph modification problems, which seems applicable to a large class of *NP*-hard problems.

## 7.1 Discussion of Results

We concentrated on the worst-case running time analysis for *NP*-hard graph modification problems. In several cases our automation framework in conjunction with relatively simple problem-specific rules yielded the so far best known upper bounds on search tree sizes for the corresponding problem. Even if our setting did not always lead to the best known worst-case bounds, however, it might be still considered scientific progress since it usually significantly reduced the "proof complexity" of the corresponding search trees when compared to the hand-made case distinctions. In this sense, our framework helps to reveal the usually few "core rules" that lie at the very heart of successfully attacking combinatorially hard problems. This may lead to a better understanding of the considered problems and may smoothen the way for new approaches in deriving smaller and smaller search tree sizes. Finally, we have shown that it is feasible to apply the basic framework not only to graph modification problems.

## 7.2 Open Problems and Challenges

Many things remain to be done.

**Improvements of the framework.** First, it would be very desirable to extend our framework in order to directly translate the computed case distinctions into "executable search tree algorithm code" and to test the thus implemented algorithms empirically.

Our approach has two main computational bottlenecks: joining of (large sets of) branching rules, and identifying the set of possible expanded windows when adding a vertex. For the first problem, recent research indicates more thorough comparison of branching vectors can keep the sets smaller; also applying heuristics seems promising. Note that heuristics will not lead to algorithms which are not exact or have unprovable exact search tree size bounds; it will rather only possibly miss some better algorithm within the set of considered algorithms.

The expansion step of finding all nonisomorphic graphs created by adding a vertex to a window is implemented very naïvely in our framework, which simply generates all possible expansions and then filters out isomorphic graphs. Better techniques are known, and available for example in the *nauty* library [McK90].

It seems like it would also be advantageous to not set a fixed limit for the maximal size of an expanded window, but rather try to work with iterative refinements of a starting algorithm, expanding further the windows which mark the "weak point" of the algorithm. This could be combined with heuristics for finding the branching rules for an expanded window, where we could "try harder" for a certain window as soon as it marks the worst case. Unfortunately, because there are usually several expansion methods for a window, it is very hard to tell whether improvements could come rather from further expansions, or rather from choosing a completely different expansion further up the expansion tree.

As with most search tree based algorithms, parallelization to reduce the high running times seems feasible.

**Improvements for the considered problems.** The possibilities of applying our framework to Cluster Editing and other graph modification problems are far from exhausted; after all, the problem-specific rules applied can be described in a few lines. Obviously, trying to develop new reduction and expansion rules may lead to improved search tree size bounds. In combination with our problem kernel reduction rules, this might yield a competitive algorithm for clustering. Eventually, we would like to test our algorithms on real data and compare it to other implementations, for example using the scoring framework by Gat-Viks, Sharan, and Shamir [GVSS03].

For most of the other considered graph modification problems, the questions for a problem kernel and a linear-time recognition method for the forbidden subgraph are still open.

**Application to other problems.** In addition, it is open to adapt our approach to other graph problems besides the considered ones or, more generally, to other combinatorial problems. The approach seems to have the potential to establish new ways for proving upper bounds on the running time of *NP*-hard combinatorial problems; we have already demonstrated this with the *NP*-hard Dominating Set problem with a maximum vertex degree of 3 in Sect. 5.3, and the application sketches in Sect. 5.5.1 and 5.5.2.

Another challenge is to use the automated framework in order to derive proofs and case distinctions simple enough that they can be verified by hand.

## 7.3 Related Work

Independently from this work, Frank Kammer and Torben Hagerup (Augsburg) informed us about ongoing related work concerning machine-generated proofs for upper bounds on hard combinatorial problems, such as Independent Set or Maximum Satisfiability. Eventually, we note that Robson [Rob01] also used the computer in order to improve the search tree of his algorithm for Independent Set [Rob86]. However, his approach seems very problem-specific and deals with special cases in his elaborate and extensive case distinction. It does not result in a general automation framework such as ours.

In recent work, Chen, Kanj, and Xia [CKX03] presented improvements on the analysis of search tree sizes. They show with amortized analysis how simple algorithms, if analyzed properly, may perform much better than suggested by upper bounds on their running time derived by considering only a worst-case scenario. They demonstrate this with a new analysis of an algorithm for Vertex Cover on degree-3 graphs, which proves a running time of $O(1.194^k + kn)$. Their approach does not provide better algorithms per se, but may lead to them when applied in algorithm design. Combining this approach with the automated algorithm design framework presented here might yield algorithms with improved search tree sizes due to the more precise selection of algorithms. It is not clear yet, though, how easy it is to generalize the approach of Chen et al. to further problems.

Two groups from St. Petersburg State University recently considered automation approaches for finding search tree algorithms for satisfiability problems. Nikolenko and Sirotkin [NS03] provide automatically derived upper bounds on the running time of propositional satisfiability (SAT) algorithms; Fedin and Kulikov [FK03] study the $(n, 3)$-MaxSAT problem.

## 7.4 Acknowledgments

I would like to thank my advisers Jens Gramm, Jiong Guo, and Rolf Niedermeier for their support and the considerable efforts they devoted to my work. They were always willing to provide me with fruitful discussions, references, insights, and tea. Also I wish to thank all my other colleagues for the enjoyable atmosphere they created at our institute.

# Bibliography

[AFF$^+$01] Jochen Alber, Hongbing Fan, Michael R. Fellows, Henning Fernau, Rolf Niedermeier, Fran Rosamond, and Ulrike Stege. Refined search tree technique for Dominating Set on planar graphs. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2001. 22, 63

[AGN01] Jochen Alber, Jens Gramm, and Rolf Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229:3–27, 2001. 11

[BBC02] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 238–247. IEEE Computer Society, 2002. 7, 19, 57

[BDSY99] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3/4):281–297, 1999. 19

[BLS99] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes: a Survey.* SIAM Monographs on Discrete Mathematics and Applications, 1999. See also *Information system on graph class inclusions*, `http://wwwteo.informatik.uni-rostock.de/isgci/`. 41

[BR99] Nikhil Bansal and Venkatesh Raman. Upper bounds for MaxSat: further improved. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC 1999)*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 1999. 6

[Cai96] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996. 7, 9, 19

[Cai03] Leizhen Cai. Parameterized complexity of Vertex Colouring. *Discrete Applied Mathematics*, 127(1):415–429, 2003. 17

[CGW03] Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. In *Proceedings of the 44rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003)*. IEEE Computer Society, 2003. To appear. 19, 57

[CK02] Jianer Chen and Iyad A. Kanj. Improved exact algorithms for MAX-SAT. In *Proceedings of the 5th Latin American Theoretical Informatics (LATIN 2002)*, volume 2286 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 2002. 6

[CKJ01] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001. 6, 10, 65

[CKX03] Jianer Chen, Iyad A. Kanj, and Ge Xia. Labeled search trees and amortized analysis: Improved upper bounds for NP-hard problems. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003)*, 2003. To appear. 77

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. English translation available at `http://caml.inria.fr/oreilly-book/`. 13

[CPS85] Derek G. Corneil, Yehoshua Perl, and Lorna K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985. 60

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999. 10, 11, 19

[DHIV01] Evgeny Dantsin, Edward A. Hirsch, Sergei Ivanov, and Maxim Vsemirnov. Algorithms for SAT and upper bounds on their complexity. Technical Report TR01-012, Electronic Colloquium on Computational Complexity, 2001. 65

[DJ02] Vilhelm Dahlhöf and Peter Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 292–298, 2002. 6

[Dow03] Rodney G. Downey. Parameterized complexity for the skeptic. In *Proceedings of the 18th IEEE Annual Conference on Computational Complexity*, pages 147–169, 2003. 11

[DP02] Limor Drori and David Peleg. Faster exact solutions for some NP-hard problems. *Theoretical Computer Science*, 287(2):473–499, 2002. 6

[Fel02] Michael R. Fellows. Parameterized complexity: The main ideas and connections to practical computing. In *Experimental Algorithmics*, volume 2547 of *Lecture Notes in Computer Science*, pages 51–77. Springer, 2002. 11

[FK03] Sergey S. Fedin and Alexander S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. Manuscript, Steklov Institute of Mathematics, St. Petersburg, September 2003. 66, 77

[FN01] Henning Fernau and Rolf Niedermeier. An efficient exact algorithm for Constraint Bipartite Vertex Cover. *Journal of Algorithms*, 38(2):374–410, 2001. 6

[GGHN03a] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Automated generation of search tree algorithms for graph modification problems. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, volume 2832 of *Lecture Notes in Computer Science*, pages 642–653. Springer, 2003. 7

[GGHN03b] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. In *Proceedings of the 5th Italian Conference on Algorithms and Complexity (CIAC 2003)*, volume 2653 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2003. 22, 24, 57, 64

[GHNR03] Jens Gramm, Edward A. Hirsch, Rolf Niedermeier, and Peter Rossmanith. Worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT. *Discrete Applied Mathematics*, 130(2):139–155, 2003. 6

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. 17, 64

[GVSS03] Irit Gat-Viks, Roded Sharan, and Ron Shamir. Scoring clustering solutions by their biological relevance. *Bioinformatics*, 2003. To appear. 76

[HHS98a] Teresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater, editors. *Domination in Graphs: Advanced Topics*, volume 209 of *Pure and Applied Mathematics.* Marcel Dekker, 1998. 62

[HHS98b] Teresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Pure and Applied Mathematics.* Marcel Dekker, 1998. 62

[Hir00] Edward A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000. 6

[HJ97] Pierre Hansen and Brigitte Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79:191–215, 1997. 18

[HK92] Lars Hagen and Andrew B. Kahng. New spectral methods for ratio cut partitioning and clustering. *EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(9):1074–1085, 1992. 18

[HK02] Edward A. Hirsch and Alexander S. Kulikov. A $2^{n/6.15}$-time algorithm for X3SAT. Technical report, Steklov Institute of Mathematics, St. Petersburg, 2002. `http://www.pdmi.ras.ru/preprint/2002/02-13.html`. 6, 65

[KM86] Mirko Křivánek and Jaroslav Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, 1986. 19

[KST99] Haim Kaplan, Ron Shamir, and Robert E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999. 20

[Kul99] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999. 6, 11

[LDG$^+$02] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system – documentation and user's manual, 2002. `http://caml.inria.fr/ocaml/htmlman/`. 13

[LVD$^+$96] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Available on the web, 1996. `http://caml.inria.fr/ocaml/`. 12, 67

[LY80] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980. 7, 18, 60

[McK90] Brendan D. McKay. **nauty** user's guide (version 1.5). Technical Report TR-CS-90-02, Australian National University, Department of Computer Science, 1990. 12, 67, 76

[MR99] Meena Mahajan and Venkatesh Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999. 6, 20

[Nat99] Assaf Natanzon. Complexity and approximation of some graph modification problems. Master's thesis, Department of Computer Science, Tel Aviv University, 1999. 19, 57

[NR99] Rolf Niedermeier and Peter Rossmanith. Upper bounds for Vertex Cover further improved. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1999)*, volume 1563 of *Lecture Notes in Computer Science*, pages 561–570. Springer, 1999. 10, 65

[NR00a] Rolf Niedermeier and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125–129, 2000. 23

[NR00b] Rolf Niedermeier and Peter Rossmanith. New upper bounds for Maximum Satisfiability. *Journal of Algorithms*, 36:63–88, 2000. 6

[NR03a] Rolf Niedermeier and Peter Rossmanith. An efficient fixed-parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 1:89–102, 2003. 6, 59, 60, 62, 64

[NR03b] Rolf Niedermeier and Peter Rossmanith. On efficient fixed-parameter algorithms for Weighted Vertex Cover. *Journal of Algorithms*, 47(2):63–77, 2003. 6, 10, 65

[NS03] Sergey I. Nikolenko and Alexander V. Sirotkin. Worst-case upper bounds for SAT: automated proof. In *15th European Summer School in Logic Language and Information (ESSLLI 2003), Student Session*, 2003. 66, 77

[NSS01] Assaf Natanzon, Ron Shamir, and Roded Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001. 7, 17, 18

[Rob86] John Michael Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986. 6, 77

[Rob01] John Michael Robson. Finding a maximum independent set in time $O(2^{n/4})$? Technical report, Université Bordeaux 1, Département d'Informatique, 2001. 6, 77

[Sch01] Uwe Schöning. New algorithms for $k$-SAT based on the local search principle. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 87–95. Springer, 2001. 65

[Sha02] Roded Sharan. *Graph Modification Problems and their Applications to Genomic Research.* PhD thesis, School of Computer Science, Tel-Aviv University, 2002. 7, 17

[SS00] Roded Sharan and Ron Shamir. CLICK: A clustering algorithm with applications to gene expression analysis. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB 2000)*, pages 307–316. AAAI Press, 2000. 19

[SS02] Ron Shamir and Roded Sharan. Algorithmic approaches to clustering gene expression data. *Current Topics in Computational Molecular Biology*, pages 269–300, 2002. 18

[SST02] Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. In *Proceedings of the 28th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2002)*, volume 2573 of *Lecture Notes in Computer Science*, pages 379–390. Springer, 2002. 7, 18, 19, 20

[WL93] Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993. 18

[Woe03] Gerhard J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Proceedings of 5th International Workshop on Combinatorial Optimization – Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–208. Springer, 2003. 5, 10