Algorithmica, Vol. 39 Number 4, pp. 321-347, Springer, 2004

Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems*

Jens Gramm† Jiong Guo‡ Falk Hüffner Rolf Niedermeier‡
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Germany
{gramm,guo,hueffner,niedermr}@informatik.uni-tuebingen.de

Abstract

We present a framework for an 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 may also lead to improved upper bounds on search tree sizes (i.e., faster exact solving algorithms) in comparison with previously developed "hand-made" search trees. Among others, such an example is given with the NP-complete Cluster Editing problem (also known as Correlation Clustering on complete unweighted graphs), which asks for the minimum number of edge additions and deletions to create a graph which is a disjoint union of cliques. The hand-made search tree for Cluster Editing had worst-case size $O(2.27^k)$, which now is improved to $O(1.92^k)$ due to our new method. (Herein, k denotes the number of edge modifications allowed.)

Keywords. NP-hard problems, graph modification, search tree algorithms, exact algorithms, automated development and analysis of algorithms, algorithm engineering.

*An extended abstract of this paper was presented at the *11th Annual European Symposium on Algorithms (ESA 2003)*, Springer-Verlag, *LNCS* 2832, pages 642–653, held in Budapest, Hungary, September 15–19, 2003.

†Supported by the Deutsche Forschungsgemeinschaft (DFG), research project OPAL (optimal solutions for hard problems in computational biology), NI 369/2.

‡Supported by the Deutsche Forschungsgemeinschaft (DFG), junior research group PIAF (fixed-parameter algorithms), NI 369/4.

1 Introduction

In the field of exactly solving NP-hard problems [1, 15, 39], often the developed algorithms employ exhaustive search based on a clever *search tree* (also called *splitting*) strategy. For instance, search tree based algorithms have been developed for Satisfiability [20, 24], Maximum Satisfiability [3, 6, 19, 30], Exact Satisfiability [12, 23], Independent Set [9, 34, 35], Vertex Cover [7, 32], Constraint Bipartite Vertex Cover [16], 3-Hitting Set [31], and numerous other problems. Moreover, most of these algorithms have undergone some kind of "evolution" towards better and better exponential-time bounds. The improved upper bounds on the running times, 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 [27] to 1.38^K [30] to 1.34^K [3] to recently 1.32^K [6], where K denotes the number of clauses in the given formula in conjunctive normal form.

Surveying algorithms for the Satisfiability problem, Dantsin et al. [10] stated that 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 paper, seemingly for the first time, we present such an automated approach for the development of efficient search tree algorithms, focusing on NP-hard graph modification problems.[1] Our work may be considered as a very special case of algorithm engineering. We present efficient programs not to solve decision or optimization problems but to develop efficient programs (i.e., search tree algorithms with "small" exponential running time). As an oddity, our tool in fact employs search trees on a meta level in order to obtain search tree algorithms as output.

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.

[1] Almost simultaneously, two groups of researchers [14, 33] obtained similar, but somewhat more special results concerning Satisfiability and a special variant of Maximum Satisfiability.

Then, the user may think of additional problem-specific rules to improve this situation, obtain a better bound, and repeat this process.[2]

The automated generation of search tree algorithms in this paper is restricted to the class of graph modification problems [4, 26, 29], 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 [36, 37] and correlation analysis in machine learning [2]:

> **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?

Note that Bansal et al. [2] and various other researchers [5, 11, 13] recently studied the approximability of the so-called Correlation Clustering on complete unweighted graphs, which is equivalent to Cluster Editing as we study here following the notation of Shamir et al. [36][3]. The currently best known polynomial-time approximation algorithm provides a factor-4 approximation [5].

Recently, we gave a search tree based algorithm exactly solving Cluster Editing in $O(2.27^k + |V|^3)$ time [18]. 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 simple problem-specific rules (whose correctness is easy to check), we obtain an $O(1.92^k + |V|^3)$ time algorithm for the same problem. It is achieved by an automated case analysis that checks much more subcases than we were able to do in [18]. 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] Clearly, not the whole process of algorithm development can be automated—human creativity (fortunately) remains an integral part that cannot be omitted in the process of development.

[3] Obviously, Bansal et al. and the others have been unaware of Shamir et al.'s work.

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 paper contributes a new way to relieve humans from awkward 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 the generation of triangle-free graphs and cographs.

2 Preliminaries

We assume familiarity with basic notations of algorithms, computational complexity, and graph theory. We only deal with undirected, simple graphs $G = (V, E)$ without self-loops. By $N(v) := \{ u \mid \{u, v\} \in E \}$ we denote the *neighborhood* of $v \in V$. We call a graph $G' = (V', E')$ *vertex-induced subgraph* of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' = \{\{u, v\} \mid u, v \in V'$ and $\{u, v\} \in E\}$. A *graph property* is a mapping from the set of graphs onto true and false.[4] Then, we can define our core problem as follows.

> Graph Modification
> **Input:** Graph G, a graph property Π, and a nonnegative integer k.
> **Question:** Is there a graph G' such that $\Pi(G')$ holds and such that we can transform G into G' by altogether at most k edge additions, edge deletions, and vertex deletions?

In this paper, we deal with special cases of Graph Modification named Edge Modification (only edge additions and deletions are allowed), Edge Deletion (only edge deletions allowed), and Vertex Deletion (only vertex deletions allowed).[5] Moreover, *hereditary* graph properties play an important role. A property Π is hereditary if for each graph G that has property Π, every vertex-induced subgraph of G also has Π. Finally, the concrete applications of our framework to be presented here refer to properties Π that have a *forbidden subgraph characterization*. For instance, consider Cluster Editing. Here, the (hereditary) property Π is "to consist of a disjoint union of cliques." It holds that this Π is true for a graph G iff G has no P_3 (i.e., a path consisting of three vertices) as a vertex-induced

[4] Note that a graph property should map two isomorphic graphs onto the same value.

[5] We do not consider Edge Completion in this paper since for cluster graphs it is trivial, for triangle-free graphs (Sect. 4.1.3) it is not meaningful, and for cographs (Sect. 4.2.2) it is equivalent to Edge Deletion.

subgraph (also cf. [18, 36]). Here, the set of forbidden subgraphs consists of one element, namely the P_3. The corresponding Edge Deletion problem is called Cluster Deletion.

In the following, whenever we mention *vertex pair*, we mean an unordered pair of distinct vertices.

Search tree algorithms. Perhaps the most natural way to organize exhaustive search is to use a search tree. For instance, consider the NP-complete Vertex Cover problem where, given a graph $G = (V, E)$ and a positive integer k, the question is whether there is a set of vertices $C \subseteq V$ with $|C| \leq k$ such that each edge in E has at least one of its two endpoints in C. The following observation gives a simple size $O(2^k)$ search tree. Consider an arbitrary edge $\{u, v\}$. Then at least one of the vertices u and v has to be in C. Thus, we can branch the recursive search into two cases, namely $u \in C$ or $v \in C$. (Note that this does *not* exclude the possibility that both u and v are in C!) Since we are looking for a set C of size at most k we easily obtain a search tree of size $O(2^k)$: The root vertex is labeled with the original graph G and k. Now one child node of the search tree is labeled with the graph that one gets when deleting u from G and the other child is labeled with the graph that one gets when deleting v from G. Moreover, both children are labeled with $k - 1$. This is done recursively until either the graph contains no more edges or we would have to spend more than k vertices to cover all edges. Both these termination conditions can easily be checked. Clearly, one can obtain better search tree sizes. For instance, we can apply the following branching rule. As long as there is a vertex with at least three neighbors in G, we branch into the following cases: Either take v or all its neighbors into C. (This is the only way to cover the edges adjacent to v!) If, however, all vertices only have one or two neighbors, then the problem is easily linear-time solvable and no exhaustive search is necessary. The search tree that derives from this branching strategy already has size at most $O(1.47^k)$, as can be determined using the mathematical tools described next.

Analysis of search tree sizes. If the algorithm solves a problem of "size" s and calls itself recursively for problems of "sizes" $s - d_1, \ldots, s - d_i$, then $(d_1, \ldots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence $t_s = t_{s-d_1} + \cdots + t_{s-d_i}$, with $t_j = 1$ for $0 \leq j < d$ and $d = \max\{d_1, \ldots, d_i\}$ (to simplify matters, without any harm, we only count the number of leaves here). Its characteristic polynomial is $z^d =$

$z^{d-d_1} + \cdots + z^{d-d_i}$ (see, e.g., Kullmann [24] for more details). We often refer to the case distinction corresponding to a branching vector $(d_1, \ldots, d_i)$ also as $(d_1, \ldots, d_i)$-*branching*. The characteristic polynomial as given here has a unique positive real root α with $t_s = O(\alpha^s)$. We call α the *branching number* that corresponds to the branching vector $(d_1, \ldots, d_i)$.

In our framework, where numerous subcases are checked automatically, an often occurring task is to *"concatenate"* branching vectors. For example, consider the two branching vector sets $S_1 = \{(1,2),(1,3,3)\}$ and $S_2 = \{(1,2,5),(2,2,2)\}$. We have to determine the best branching vector when concatenating every element of S_1 with every element of S_2. In the concrete example, we obtain four possibilities. It is important to note that we cannot simply take the best branching vector from S_1 and concatenate it with the best one from S_2. In our example, in S_1, branching vector $(1,2)$ (branching number 1.62) is better than branching vector $(1,3,3)$ (branching number 1.70), and in S_2, branching vector $(1,2,5)$ (branching number 1.71) is better than branching vector $(2,2,2)$ (branching number 1.74); however, the concatenation of the two best branching vectors $(1,2,1,2,5)$ with a branching number of 2.75 is not optimal; in fact, concatenating the two worst branching vectors, leading to the branching vector $(1,3,3,2,2,2)$ with a branching number of 2.52, is the best choice.

Since in our applications the sets S_1 and S_2 can generally get rather large, it would save much time not having to check every pair of concatenations. We use the following simplification. Consider a branching vector as a multi-set of its entries, 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 element $x \in b_1$ 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 a better solution no matter what we concatenate to it. In the current setting, we only used this criterion when pruning branching vectors. Since the concatenation process is one of the most time-consuming parts of our framework, further improvements to speed up this process are under current development.

3 The General Technique

Search tree algorithms basically consist of a set of branching rules. Branching rules are usually based on local substructures. For graph problems, these

can be induced subgraphs having up to s vertices for a constant integer s; we refer to graphs having s vertices as *size-s graphs*. Then, each branching rule specifies the branching for a particular local substructure. The idea behind our automation approach is roughly described as follows:

(1) For constant s, enumerate all "relevant" subgraphs of size s such that every input instance of the given graph problem has s vertices inducing at least one of the enumerated subgraphs.

(2) For every local substructure enumerated in Step (1), check all possible branching rules for this local substructure and select the one corresponding to the *best*, i.e., smallest, branching number. The set of all these best branching rules then defines our search tree algorithm.

(3) Determine the worst-case branching rule among the branching rules stored in Step (2), because this branching rule yields a worst-case bound on the search tree size of the generated search tree algorithm.

Note that both in Step (1) and Step (2), we usually make use of further *problem-specific rules*: For example, in Step (1), problem-specific rules can determine input instances which do not need to be considered in our enumeration, e.g., instances which can be solved in polynomial time, instances which can be simplified due to reduction rules, or instances for which we can use a manually developed branching rule; instances with one of these properties are referred to as "trivial" instances. In this section, we restrict ourselves to describing a general framework, indicating where problem-specific rules may apply. The problem-specific rules corresponding to particular graph modification problems are, then, given in the following sections.

In the next two subsections, we discuss Steps (2) and (1), respectively, in more detail. We will use Cluster Deletion as a running example. Cluster Deletion is an Edge Deletion problem in which the forbidden induced subgraph is a P_3, i.e., a path consisting of three vertices.

3.1 Computing a Branching Rule for a Subgraph

We outline a general framework to generate, given a size-s graph $G_s = (V_s, E_s)$[6] for constant s, an "optimal" branching rule for G_s. 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*.

[6] We assume that the vertices of the graph are ordered and that we can write $u < v$ or $v < u$ for $u, v \in V_s$ with $u \neq v$.

We describe our framework for the example of Cluster Deletion. However, this framework is suitable also for other graph modification problems and also for graph problems in general; at the end of this subsection, we indicate where problem-specific changes have to be made in the framework. Our central reference point in this subsection is the meta search tree procedure `compute_br()` given in Fig. 1. In the following paragraphs we describe `compute_br()` in a step-by-step manner.

(1) Branching rules and branching objects. A branching rule for G_s specifies a set of "simplified" (to be made precise in the next paragraph) graphs $G_{s,1}, G_{s,2}, \ldots, G_{s,r}$. When invoking the branching rule, one would replace, for every $G_{s,i}$, $1 \leq i \leq r$, G_s by $G_{s,i}$ and invoke the search tree procedure recursively on the thereby generated instances. By definition, the branching rule has to satisfy the following property: a "solution" is an optimal solution for G_s iff it is "best" among the optimal solutions for all $G_{s,i}$, $1 \leq i \leq r$, produced by the branching rule. This is referred to by saying that the branching rule is *complete*. The *branching objects* are the objects on which the branching rule to be constructed branches; the branching objects are determined depending on the particular problem. In Cluster Deletion, the branching objects are the vertex pairs of G_s since we obtain a solution graph by deleting edges.

(2) Annotations. A "simplified" graph $G_{s,i}$, $1 \leq i \leq r$, is obtained from G_s by assigning labels to a subset of branching objects in G_s. The employed set of labels is problem-specific. Depending on the problem, certain branching objects may also initially carry "problem-inherent" labels which cannot be modified by the meta search tree procedure. An *annotation* is a partial mapping π from the branching objects to the set of labels; if no label is assigned to a branching object then π maps to "undef." Let π and π' both be annotations for G_s, then π' *refines* π iff, for every branching object b, it holds that

$$\pi(b) \neq \text{undef} \implies \pi'(b) = \pi(b).$$

As to Cluster Deletion, the labels for a vertex pair $u, v \in V_s$ can be chosen as *permanent* (i.e., the edge is in the solution graph to be constructed) or *forbidden* (i.e., the edge is not in the solution graph to be constructed). In Cluster Deletion, all vertex pairs sharing no edge are initially assigned the label *forbidden* since edges cannot be added; these are the problem-inherent labels. By G_s with annotation π, we, then, refer to the graph

Procedure `compute_br(`π`)`
Global: Graph $G_s = (V_s, E_s)$.
Input: Annotation π for G_s (for a definition of annotations see paragraph (2)).
Output: Set B of branching rules for G_s with annotation π.

Method:

```
B := ∅;                    /* set of branching rules, to be computed */
π:=br_reduce(π);                                          /* (4) */
for all {u,v} ∈ E_s with u < v and π(u,v) = undef do
    π_1:=π;
    π_1(u,v):=permanent;     /* annotate edge as permanent (5) */
    B_1:=compute_br(π_1);
    π_2:=π;
    π_2(u,v):=forbidden;     /* annotate edge as forbidden (5) */
    B_2:=compute_br(π_2);
    B:=B ∪ br_concatenate(B_1, B_2);        /* concatenating and
                                  pruning branching rules (5) */
endfor;
if num_edge_mod(π) > 0 then B:=B ∪ {π}; endif;          /* (5) */
return B;
```

Figure 1: Meta search tree procedure for Cluster Deletion in pseudocode. Numbers in comments refer to the explaining paragraphs in Sect. 3.1

obtained from G_s by deleting $\{u, v\} \in E_s$ if π assigns the label *forbidden* to (u, v). Thus, an annotation specifies, in particular, a set of edges to delete from the input graph. In this way, an annotation can be used to specify one branch of a branching rule.

(3) Representation of branching rules. A branching rule for G_s with annotation π can be represented by a set A of annotations for G_s such that, for every $\pi' \in A$, π' refines π. Then, every $\pi' \in A$ specifies one branch of the branching rule. A set A of annotations has to satisfy the following three conditions in order to specify a branching rule: (a) The branching rule is complete. (b) Every annotation decreases the *search tree measure*, i.e., the parameter with respect to which we intend to measure the search tree size.

(c) The subgraph consisting of the annotated branching objects has to fulfill every property required for a solution of the considered graph problem.

In Cluster Deletion, the search tree measure is the number of edge deletions. Therefore, condition (b) implies that every annotation deletes at least one edge from the graph. Condition (c) means that the annotated vertex pairs do not form a P_3, i.e., there are no $u, v, w \in V_s$ with $\pi(u, v) = \pi(v, w) =$ permanent and $\pi(u, w) =$ forbidden.

(4) Problem-specific rules that refine annotations. To obtain nontrivial bounds it is decisive to have a set of problem-specific *reduction rules*. In our terminology, a reduction rule specifies how to refine a given annotation π to π' such that an optimal solution for the input graph with annotation π' is also an optimal solution for the input graph with annotation π.

For Cluster Deletion, we have the following reduction rule (for details see Sect. 4):

> Reduction Rule: Given a graph $G = (V, E)$ with annotation π, if there are three pairwise distinct vertices $u, v, w \in V$ with $\pi(u, v) = \pi(v, w) =$ permanent, then we can replace π by an annotation π' which refines π by setting $\pi'(u, w) :=$ permanent. Analogously, if $\pi(u, v) =$ permanent and $\pi(v, w) =$ forbidden, then $\pi'(u, w) :=$ forbidden.

In Fig. 1, the reduction rule is implemented by procedure `br_reduce(`π`)` which receives as input an annotation π and returns a refined annotation π' which is obtained by applying the reduction rule to π.

(5) Meta search tree. Procedure `compute_br()`, for graph G_s, has as input an annotation π. It returns a set of possible branching rules for G_s with annotation π, i.e., a set $B = \{A_1, \ldots, A_r\}$ of annotation sets such that, for every $1 \leq i \leq r$ and every $\pi' \in A_i$, π' refines π. Each $\pi' \in A_i$ represents one branch of the branching rule given by A_i.

Given G_s with an annotation π, we choose, if there is one, a non-annotated branching object and, for every possible label, recursively consider the subcase in which the branching object is assigned the label. (If all branching objects are annotated then the recursion stops.) In the ith subcase, we receive a set B_i of branching rules. One way to obtain a valid branching rule for G_s with annotation π is the concatenation of branching rules from all subcases, taking one branching rule from every subcase; the concatenation of branching rules is described in Sect. 2. Implicitly, we omit, during the concatenation, branching rules which have a branching vector

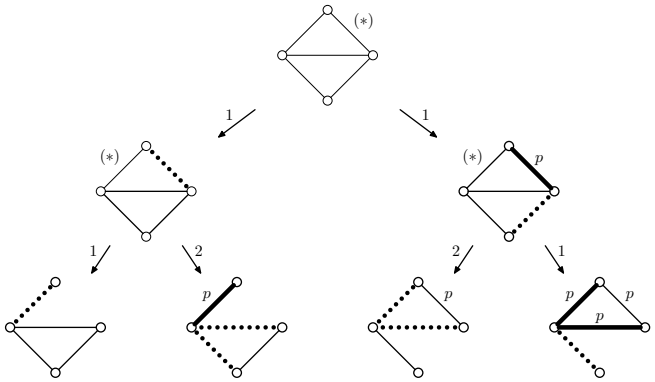

Figure 2: Illustration of a meta search tree traversal for Cluster Deletion. At the root we have a size-4 input graph having no labels. *Arrows* indicate the branching steps of the meta search tree. We only display branches of the meta search tree which contribute to the computed branching rule. The vertex pair on which we branch is indicated by $(*)$. Permanent edges are indicated by p, vertex pairs sharing no edge are implicitly forbidden (*bold* or *dotted lines* indicate when a vertex pair is newly set to permanent or forbidden, respectively). Besides the vertex pair on which we branch, additional vertex pairs are set to permanent or forbidden due to the problem-specific reduction rule explained in Sect. 3.1(4). The numbers at the arrows indicate the number of edges deleted in the respective branching step. Thus, the resulting branching rule is determined by the leaves of this tree and the corresponding branching vector is $(2, 3, 3, 2)$

being subsumed by the branching vector of another rule in the same set (see Sect. 2). A second way to obtain a valid branching rule for G_s with annotation π can be to take $\{\pi\}$, as a rule having only one branch; however, $\{\pi\}$ is only a valid branching rule if (see condition (b) stated in paragraph (3)) π already implies a decrease of the search tree measure.

The root of the meta search tree is, for a non-annotated input graph, given by calling `compute_br(`π_0`)` with G_s and an annotation π_0 which assigns the problem-inherent labels to some branching objects (e.g., in Cluster Deletion the forbidden labels to vertex pairs sharing no edge) and maps all other branching objects to undef. The call `compute_br(`π_0`)` results in a set B of possible branching rules. From B, we select the *best* branching rule (with smallest branching number).[7]

[7] In the resulting search tree algorithm, we assume, in general, an input graph in which

In Cluster Deletion, a branching rule is computed for every non-annotated vertex pair. Two subcases are considered, one in which the vertex pair is set to permanent and one in which the vertex pair is set to forbidden. If π already implies edge deletions for G_s (in Fig. 1, the number of edges to be deleted is determined by `num_edge_mod`(π)) then π itself is a branching rule for G_s with annotation π. In Fig. 2, we illustrate the meta search tree generated for Cluster Deletion on a size-4 graph.

(6) Storing already computed branching rules. When branching as described in (5), it is possible that the meta search tree procedure is invoked for annotations π for which it was already invoked before. For example, a situation in which two branching objects are annotated is reached by annotating one object and then the other but also by annotating them in reversed order. To avoid this, we store an annotation which has already been processed together with its computed set of branching rules.

(7) Generalizing the framework. In this section, we concentrated on the Cluster Deletion problem. We claim, however, that this framework is usable for graph problems in general. Two main issues where changes have to be made depending on the considered problem:

- In Cluster Deletion, the branching objects are vertex pairs and the possible labels are "permanent" and "forbidden." In general, the branching objects and an appropriate set of labels for them are determined by the considered graph problem. For example, in Vertex Cover, the objects to branch on are vertices instead of edges and, thus, the labels would be assigned to the vertices. The labels would be, e.g., "is in the vertex cover" and "is not in the vertex cover." Depending on the problem, additional auxiliary labels might be helpful, e.g., reflecting the vertex degree.

- The reduction rules are problem-specific. To design an appropriate set of reduction rules working on local substructures probably is the most challenging part when applying our framework to a new problem and for the development of practical search tree algorithms in general.

the branching objects do not carry labels. The labels assigned to branching objects by a branching rule can and should, however, be kept after the application of a branching rule. When applying a branching rule to a graph in which branching objects already carry labels, an annotation is not allowed to "conflict" with these labels; branches in which the annotation would change a label can simply be omitted.

In this subsection, we presented the meta search tree procedure for the example of Cluster Deletion. As input it takes a local substructure and a set of reduction rules. It explores the set of *all possible* branching rules on this local substructure, taking into account the given reduction rules. Therefore, this yields the following theorem where π_0 denotes the annotation assigning the problem-inherent labels of Cluster Deletion:

Theorem 1. *Given a graph G_s with s vertices for constant s, the set of branching rules returned by* `compute_br(`π_0`)` *(Fig. 1) contains, for this fixed s, a best branching rule for* Cluster Deletion *among those that can be obtained by branching only on vertex pairs from G_s and by only using the reduction rules performed by* `br_reduce()`*.*

3.2 A More Sophisticated Enumeration of Local Substructures

In the introduction of Sect. 3, we described how to compute branching rules for a given graph problem: We enumerate, given a constant integer s, all non-isomorphic size-s graphs; for each of these graphs, we compute a branching rule as described in Sect. 3.1. Using problem-specific rules, we can restrict the enumeration to a set of non-isomorphic size-s graphs such that every non-trivial input instance contains at least one of the enumerated graphs as a vertex-induced subgraph. This will lead to considerable improvements concerning the worst-case branching rules and concerning the running time of our automated generation.

Example. Considering the Cluster Deletion problem, we can use the following problem-specific rules improving the enumeration of graphs: Given a constant integer s, we can, firstly, assume that every connected component in a given instance has at least s vertices; every connected component of the input graph having less than s vertices can be processed in constant time. Therefore, we can restrict the enumeration to connected size-s graphs. Secondly, we know that a non-trivial Cluster Deletion instance contains a P_3 as a vertex-induced subgraph since, otherwise, the input graph is already a solution. Therefore, we can restrict the enumeration to connected size-s graphs having a P_3 as a vertex-induced subgraph. □

Graph expansion. In the following, we show how to refine the enumeration strategy further, using problem-specific rules, in order to find a set of branching rules leading to better worst-case bounds on the size of the generated search tree. For this purpose, we introduce the concept of an *expansion* for a given size-i graph G with $1 \leq i < s$. Given a particular graph

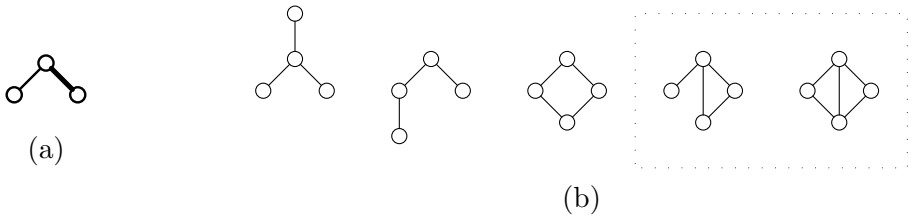

Figure 3: Illustrating the expansion of a size-three graph to a size-four graph: (a) P_3 graph, (b) the graphs to which the P_3 is expanded, i.e., all non-isomorphic graphs obtained by adding a vertex to the P_3 and by connecting this vertex in an arbitrary way by at least one edge to the vertices of the given P_3. The *dotted box* indicates the graphs in an improved expansion due to the problem-specific "common neighbor" rule for Cluster Deletion: endpoints of the *bold edge* must have a common neighbor" (see Example)

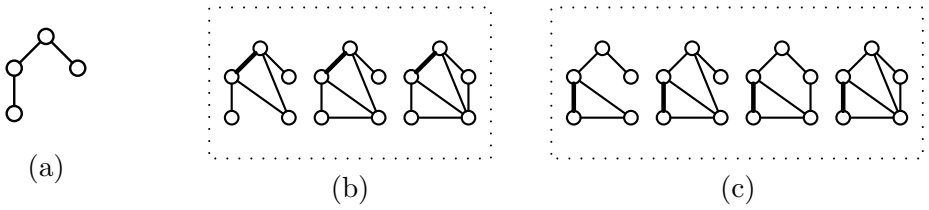

Figure 4: Two non-trivial expansions, shown in (b) and (c), for the graph shown in (a), based on the "common neighbor" rule for Cluster Deletion. By *bold lines*, we indicate the edge for which the rule was applied to obtain the expansion.

problem, an *expansion* of a size-i graph G is a set S_G of graphs of size $> i$ such that every input graph containing G as a vertex-induced subgraph also contains one of the graphs in S_G as a vertex-induced subgraph; we *expand* G by computing an expansion for G. The trivial expansion of G is to add a vertex to G and to consider all possible ways to connect the new vertex by edges with the vertices in G.

Example (continued). Because of the outlined problem-specific rules for Cluster Deletion, we can restrict our enumeration of vertex-induced subgraphs of size 3 to P_3 (shown in Fig. 3(a)). When expanding a P_3 to a set of graphs having four vertices, we obtain the graphs shown in Fig. 3(b). □

Searching the "best" expansion. Instead of expanding G in the trivial way to a set S_G, problem-specific rules can give better ways to expand a graph to only a subset of S_G.

Example (continued). As will be formally proven in Sect. 4, for Cluster Deletion, we can state the following "common neighbor rule": For an edge $\{u, v\}$ in the input graph, we can assume that u and v have a common neighbor; otherwise a "good" branching rule is known. This gives us the possibility to expand a P_3 (shown in Fig. 3(a)) only to the set of graphs indicated by the dotted box in Fig. 3(b). Figure 4 illustrates that this rule can lead (for the graph shown in (a)) to different non-trivial expansions (shown in (b) and (c)). □

The problem-specific rules might lead to several possible expansions of a given size-i graph G. To give a branching rule for G, it is sufficient to take the union of branching rules over all graphs in an expansion of G. The branching number of an expansion is determined by the largest branching numbers among these branching rules. From all possible expansions of G, however, we can select the one leading to the "best" set of branching rules. Therefore, we choose, over all known expansions, an expansion with a minimum branching number.

In Fig. 5, we show in pseudocode how to implement this improved enumeration strategy in a recursive way, returning a branching rule based on local subgraphs having at most s vertices. We represent a branching rule for a graph problem by a set of triples (G, A, p) where G denotes a graph of size at most s, A denotes the annotation set specifying a branching rule for G, and p denotes the branching number for the branching rule in A. In the line marked by (1) in Fig. 5, we iterate over all expansions of G. In (2), we successively generate the union of branching rules over all graphs in an expansion of G, by recursively calling the procedure for every graph

Procedure `graph_enumerate(`G`)`

Global: Positive integer s denoting the size, i.e., number of vertices, of the graphs to be enumerated.

Input: Graph $G = (V, E)$ with $|V| \leq s$.

Output: Branching rule C represented as a set of triples (G', A, p), where G' is a graph having at most s vertices, A is a set of annotations for G' (representing the branching rule for G'), and a real number $p \geq 1$ denoting the branching number for the branching rule in A for G'.

Method:

```
/* Compute the best branching rule for graph G */
A := compute a set of annotations specifying the best
     branching rule for G as explained in Sect. 3.1(5);
p := branching number of branching rule given by A;
C_best := {(G, A, p)}; p_best := p;
if |V| < s then
    /* Graph G has size less than s -> expand G */
    for every expansion S_G of G do                          /* (1) */
        C_{S_G} := ∅;
        /* For every graph in the expansion,
           determine the best branching rule */
        for every G' ∈ S_G do
            C_{S_G} := C_{S_G} ∪ graph_enumerate(G');        /* (2) */
        endfor;
        /* Among all branching rules for the expansion,
           determine the worst-case branching rule */
        p_{S_G} := max{p | (G, A, p) ∈ C_{S_G}};             /* (3) */
    endfor;
    /* Among all expansions of G, determine the expansion
       with the best branching number */
    (C_best, p_best) := select (C_{S_G}, p_{S_G}) from all expansions S_G
                        of G such that p_{S_G} is minimized; /* (4) */
endif;
return C_best;
```

Figure 5: Pseudocode for procedure `graph_enumerate(`G`)`. Numbers refer to explanations in Sect. 3.2

in the expansion. In (3), we determine, for every considered expansion, its branching number. In (4), we select, from all possible expansions of G, the one leading to the "best" set of branching rules. The recursion stops if G already has reached size s. Implicitly, we assume that the processed graphs are stored with the branching rules computed for them since we may encounter isomorphic graphs several times in our search, e.g., since a size-$(i+1)$ graph can belong to several expansions of a size-i graph. Initially, the procedure given in Fig. 5 is, for a constant integer s, invoked by `graph_enumerate(`G_0`)` where G_0 is a graph containing only one vertex.

Cutoff values. In the application, we can speed-up the search by user-defined cutoff values for the branching number: If a size-i graph G already yields a branching number better than the cutoff value, we omit to expand G. The application of cutoff values makes sense when one is satisfied with a certain search tree size or when we are mainly interested in the worst-case bound on the search tree size and when there is little hope for improving this worst-case bound below some "threshold value."

4 Applications and Results

The developed software consists of about 1900 lines of Objective Caml [25] code and 1500 lines of low-level C code for the graph representation, which uses simple bit vectors. The generation of canonical representations of graphs (for isomorphism tests and hash table operations) is done by the *nauty* library [28]. Branching vector sets are represented as tries, which allow for efficient implementation of the subsumption rules presented in Sect. 2.

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

We applied the general technique to 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. These will be referred to in the tables to follow.

size: Maximum number of vertices in the local subgraphs considered.

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 on the size of the resulting search tree).

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 then give the average size of the employed search trees.

bvmax: Maximum length of a branching vector occurring in the computed set of branching rules; this gives a measure for the intricacy of the generated search tree algorithm.

bvmed: 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).

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

4.1 Edge Modification Problems

Edge modification problems recently have attracted considerable interest [29, 37]. We will consider three of these in our context.

4.1.1 Application to Cluster Editing

The problem. Cluster Editing is defined as follows:

> **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 vertex-disjoint union of cliques?

As mentioned in Sect. 2, a graph consisting of a vertex-disjoint union of cliques contains no P_3 (path of three vertices) as a vertex-induced subgraph. Cluster Editing is NP-complete [22, 36, 2] and it has been used for the clustering of gene expression data [38]. As already mentioned in Sect. 1, we gave in [18] an exact algorithm for this problem based on a bounded search tree of size $O(2.27^k)$, where k denotes the allowed number of edge additions and edge deletions.

A basic search tree approach is described as follows for a given input graph $G = (V, E)$. For each P_3, i.e., $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$, the algorithm either deletes $\{u, v\}$, or deletes $\{u, w\}$, or adds $\{v, w\}$ and processes each of these cases recursively. If an algorithm can destroy all vertex-induced P_3's after at most k edge modifications then the given instance has a solution. It is easy to see that the size of the resulting search tree is upperbounded by $O(3^k)$. With more refined case distinctions the size of the search tree can be reduced to $O(2.27^k)$ as described in [18]. However, a better bound can be achieved by our automated approach.

Problem-specific rules. Following the general scenario from Sect. 3, we make use of problem-specific rules in our framework for an input graph $G = (V, E)$. We use the same labels "forbidden" and "permanent" as for Cluster Deletion.

> **Rule 1:** While enumerating subgraphs, we consider only connected instances containing a P_3 as a vertex-induced subgraph.

The correctness of Rule 1 is obvious.

As introduced in Sect. 3.1 for Cluster Deletion, the algorithm assigns labels "permanent" and "forbidden" to vertex pairs. Here, a vertex pair (u, v) annotated as "permanent" means that in the solution graph there is an edge between u and v and this edge may not be deleted; annotation "forbidden" means that in the solution graph there is no edge between the two vertices.

> **Rule 2:** For vertices $u, v, w \in V$ such that both (u, v) and (v, w) are annotated as permanent, we annotate also vertex pair (u, w) as permanent; if (u, v) is annotated as permanent and (v, w) as forbidden, then we annotate (u, w) as forbidden.

The correctness of Rule 2 follows from the requirement that a solution graph does not contain a P_3 as a vertex-induced subgraph.

The third rule allows non-trivial graph expansions (see Sect. 3.2).

Rule 3: In a non-trivial input graph, for every edge $\{u, v\} \in E$, u and v have a common neighbor.

Rule 3 is based on the following proposition which allows us to apply a "good" branching rule if there is an edge whose endpoints have no common neighbor:

Proposition 1. *Given an input graph $G = (V, E)$ for* Cluster Editing. *If there is an edge $\{u, v\} \in E$ where u and v have no common neighbor and $|(N(u) \cup N(v)) \setminus \{u, v\}| \geq 1$, then we can apply a branching rule giving a branching vector $(1, 2)$.*

Proof. We firstly consider the most simple case, $|(N(u) \cup N(v)) \setminus \{u, v\}| = 1$, and assume that $N(u) \setminus \{v\} = \{x\}$. It is easy to observe that deleting $\{u, x\}$ is always an optimal choice to destroy the P_3 formed by u, v, x regardless of the rest graph.
In the case $|(N(u) \cup N(v)) \setminus \{u, v\}| = 2$, we firstly assume that u has two "private" neighbors $x \neq y$, $x \neq v$, $y \neq v$, $\{v, x\} \notin E$, and $\{v, y\} \notin E$. Then, we have two induced P_3's, v, u, x and v, u, y. There are two cases to distinguish. The first case is that u and v are not in the same clique of the final cluster graph, for which we have to delete $\{u, v\}$. The second case is that u and v are in the same clique. Furthermore, x or y can also be in this clique. However, we have to make at least two edge modifications, namely, to insert $\{v, x\}$ and $\{v, y\}$ in order to include x and y in this clique, or to insert $\{v, x\}$ and to delete $\{u, y\}$ or vice versa in order to include only one of x and y in this clique, or to delete both $\{u, x\}$ and $\{u, y\}$ in order to exclude x and y from this clique. Therefore, we can conclude that deleting $\{u, x\}$ and $\{u, y\}$ is always an optimal choice regardless of the rest graph. Summarizing the two cases, we can achieve a $(1, 2)$-branching.

If each of u and v has a private neighbor, i.e., u has a neighbor $x \neq v$ and v has a neighbor $y \neq u$, we can make a similar case distinction as above such that a $(1, 2)$-branching can be achieved.

It is easy to prove that we can achieve an even better branching if u and v have more than two private neighbors. □

Given a Cluster Editing instance $(G = (V, E), k)$, we can apply the branching rule described in Proposition 1 as long as we find an edge satisfying the conditions of Proposition 1; the resulting graphs are called reduced with respect to Rule 3. If we already needed more than k edge modifications before the graph is reduced with respect to Rule 3 then we reject it.

Table 1: Results for Cluster Editing: (1) Enumerating all size-s graphs containing a P_3; (2) Expansion scheme additionally utilizing Rule 3

	size	time	isom	concat	graphs	maxbn	avgbn	bvmax	bvmed	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

Based on Proposition 1, Rule 3 can reduce the size of an expansion S_G for a given size-i graph G. Moreover, if G contains more than one edge whose endpoints have no common neighbor then we can have more than one different non-trivial expansion for G. However, as stated in Sect. 3.2, we have, then, the choice to select the expansion leading to the "best" set of branching rules, i.e., an expansion with a minimum branching number. If G contains no such edge, then we expand G in the trivial way, i.e., we consider all possible ways to connect the new vertex by edges with the vertices in G.

Results. See Table 1. The measured values are defined in the beginning of Sect. 4.

Only using Rules 1 and 2, we obtain the worst-case branching number 2.16 when considering induced subgraphs containing six vertices. We observe a decrease in the computed worst-case branching number maxbn with every increase in the sizes of the considered subgraphs. These results support the conjecture that the program could compute better branching rules when considering subgraphs with more vertices. However, the running time of our program increases as the graph size increases. The typical number of case distinctions for a subgraph (bvmed) seems high compared to human-made case distinctions, but should pose no problem for an implementation (which might also be done by an automated framework).

When additionally using Rule 3, we use the expansion approach presented in Sect. 3.2. In this way, we can decrease maxbn to 1.92. 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 1.80 for the computed set of branching rules is significantly lower than the worst-case one.

The main reason for the high running times is that the case distinction

in the meta search tree 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 1). As we have stated in Sect. 2, besides the subsumption mechanism, we have not applied methods to efficiently determine the best concatenation of two sets of branching vectors other than basically trying all possibilities. It can be observed from Table 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). An obvious strategy to apply here might be the use of heuristic concatenation rules. The high cost of branching vector concatenation is also the reason that graph isomorphism testing, perhaps surprisingly, contributes a decreasing proportion (isom) to the running time when increasing the graph size.

In Appendix A, we provide the expansion steps and the generated search tree for one example graph which is expanded. Summarizing the results together with previous work [18] (where a search tree size of $O(2.27^k)$ instead of $O(1.92^k)$ is given), we have the following theorem:

Theorem 2. 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 [2]. The best known polynomial-time approximation algorithm for minimizing the number of of edge modifications yields an approximation factor of only 4 [5], giving particular importance to exact fixed-parameter algorithms for this problem.

4.1.2 Application to Cluster Deletion

The problem. This is the special case of Cluster Editing where only edge deletions are allowed and is defined as follows:

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

Cluster Deletion is NP-complete [36]. In [18], we gave an algorithm for Cluster Deletion with search tree size $O(1.77^k)$.

Problem-specific rules. Since this problem is a special case of Cluster Editing, where only edge deletions are allowed, all problem-specific rules devised for Cluster Editing can also be used for this problem without any modification. However, the first implementation with all rules for Cluster Editing showed that (as shown in the first half of Table 2) the resulting worst-case branching number 1.62 is determined by the $(1, 2)$-branching of Proposition 1 which is used in Rule 3. In order to achieve a better branching rule for Cluster Deletion, we improved the branching in Proposition 1 as follows:

Proposition 2. *Given an input graph $G = (V, E)$ for* Cluster Deletion. *If there is an edge $\{u, v\} \in E$, where u and v have no common neighbor and $|(N(u) \cup N(v)) \setminus \{u, v\}| \geq 1$, then we can apply a branching rule giving a branching vector $(1, 3)$.*

Proof. As shown in the proof of Proposition 1, there is no need for branching if $|(N(u) \cup N(v)) \setminus \{u, v\}| = 1$. If $|(N(u) \cup N(v)) \setminus \{u, v\}| > 2$, then we have at least a $(1, 3)$-branching because we have to delete either $\{u, v\}$ or all other edges adjacent to u and v. For the case that $|(N(u) \cup N(v)) \setminus \{u, v\}| = 2$, we distinguish two cases:

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

   (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. Hence, no branching occurs.

   (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; otherwise, u, v, x, y form an isolated component and we can destroy all vertex-induced P_3's in this component by deleting $\{u, v\}$. 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 a private neighbor, i.e., $x \in N(u)$ and $y \in N(v)$, $x \neq y$:

(2.1) If one of x and y has no neighbor besides u and v, then we delete $\{u, v\}$. Assume that y has no neighbor besides v. For the case that x and u are in the same clique of the final cluster graph, 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 different from u, then we consider the edge $\{u, x\}$. Since u and x have no common neighbor and $|(N(u) \cup N(x)) \setminus \{u, x\}| > 2$, 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 that v and y are in the same clique of the final cluster graph. For this case, we have to delete the edges 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. □

Results. See Table 2. The measured values are defined in the beginning of Sect. 4. Incorporating Proposition 2 into Rule 3, we obtained the results shown in the second half of Table 2. Here, the $(1, 3)$-branching in Proposition 2, which corresponds to a branching number of 1.47, is not the worst case any more. These results demonstrate the benefit of interaction between manual design of problem-specific rules and mechanized case analysis.

Summarizing these results together with previous work [18] (where a search tree size of $O(1.77^k)$ instead of $O(1.53^k)$ is shown), we have the following theorem:

Theorem 3. Cluster Deletion *can be solved in* $O(1.53^k + |V|^3)$ *time.* □

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

	size	time	isom	concat	graphs	maxbn	avgbn	bvmax	bvmed	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

4.1.3 Application to Triangle Edge Deletion

The problem. Triangle Edge Deletion is defined as follows:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Can we transform G, by deleting at most k edges, into a graph that contains no triangle as a vertex-induced subgraph?

Triangle Edge Deletion can be reduced to 3-Hitting Set. The NP-complete d-Hitting Set problem (cf. [17]) is defined as follows:

> **Input:** A collection C of subsets of size at most 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?

We translate edges of the given graph to elements of the base set in the 3-Hitting Set instance. Furthermore, each triangle in the given graph translates into a three-element subset of the base set, thereby forming, together with the unchanged parameter k, an instance of 3-Hitting Set.

Problem-specific rules. Analogous rules to Rules 1 and 2 for Cluster Editing can also be found for this problem. In addition, regarding the expansions, we only sketch the respective rule used here: We assume that every edge being part of a triangle is part of at least two triangles. If there is an edge $\{u, v\} \in E$ being part of only one triangle $\{u, v\}$, $\{v, w\}$, $\{u, w\} \in E$, then this yields a $(1, 1)$-branching, in one branch deleting $\{v, w\}$ and in the other branch deleting $\{u, w\}$. It is straightforward to show that it is never better to delete $\{u, v\}$. Details are omitted.

Table 3: Results for Triangle Edge Deletion: (1) Expansion scheme utilizing the specific expansion rule given in Sect. 4.1.3

	size	time	isom	concat	graphs	maxbn	avgbn	bvmax	bvmed	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

Results. See Table 3. This problem is an example where the mechanized analysis so far could not improve an existing search tree algorithm. Since Triangle Edge Deletion can be reduced to 3-Hitting Set, we can solve it using an elaborated, hand-made search tree algorithm for 3-Hitting Set having a worst-case branching number of 2.27 [31], whereas the worst-case branching number determined by our analysis is 2.47 when considering graphs of 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.

4.2 Vertex Deletion Problems

We have also applied our automated approach to three Vertex Deletion problems that have a forbidden subgraph characterization. The problems studied in this subsection belong to the class of so-called *vertex deletion problems for hereditary properties*. As shown by Lewis and Yannakakis [26], basically all these problems are NP-complete.

Vertex Deletion problems where the graph property is given by a forbidden vertex-induced subgraph of size $d > 1$ can be reduced to the NP-complete d-Hitting Set problem.

Lemma 1. *Given a* Vertex Deletion *problem where the graph property is given by a forbidden vertex-induced subgraph of size d, then there is an $O(n^d)$ time many-one reduction to d-*Hitting Set*. Herein, n denotes the number of graph vertices.*

Proof. For a given instance of a Vertex Deletion problem 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 vertex-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 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. □

4.2.1 Application to Cluster Vertex Deletion and Triangle Vertex Deletion

The problems.

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question in the case of** Cluster Vertex Deletion**:** Can we transform G, by deleting at most k vertices, into a set of disjoint cliques?
> **Question in the case of** Triangle Vertex Deletion**:** Can we transform G, by deleting at most k vertices, into a graph that contains no triangle as vertex-induced subgraph?

Each of these two graph problems specifies a forbidden vertex-induced subgraph of three vertices, i.e., an induced P_3 or an induced K_3, respectively. As outlined above, the problems, therefore, can be reduced to the NP-complete 3-Hitting Set. For 3-Hitting Set, an elaborate search tree algorithm has been given with $O(2.27^k + |C|)$ running time [31]. A consequence of this reduction is that the described Vertex Deletion problems can also be solved by the search tree algorithm for 3-Hitting Set.

Problem-specific rules. The problem-specific rules applied here are analogous to Rules 1 and 2 given for Cluster Editing. Additionally, we use a special expansion rule in the case of Triangle Vertex Deletion: We assume that every vertex being part of a triangle is part of at least two triangles. If there is a vertex $u \in V$ being part of only one triangle $\{u, v\}$, $\{v, w\}$, $\{u, w\} \in E$, then this yields a $(1, 1)$-branching, in one branch deleting vertex v and in the other branch deleting vertex w. It is straightforward to show that it is never better to delete u. Details are omitted.

Table 4: Results for Cluster Vertex Deletion: (1) Enumerating all size-s graphs containing a P_3; (2) Expansion scheme with cutoff (see Sect. 3.2)

	size	time	isom	concat	graphs	maxbn	avgbn	bvmax	bvmed	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: Results for Triangle Vertex Deletion: (1) Expansion scheme utilizing problem-specific expansion rule; (2) additionally using cutoff values

	size	time	isom	concat	graphs	maxbn	avgbn	bvmax	bvmed	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

Results. See Table 4 for Cluster Vertex Deletion and Table 5 for Triangle Vertex Deletion. Using the enumeration without non-trivial 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 [31]. Using a cutoff value 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 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 and applying it in practice, however, a better average branching number might be more desirable than a better worst-case branching number.

4.2.2 Application to Cograph Vertex Deletion

The problem.

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Can we transform G, by deleting at most k vertices, into a cograph, i.e., into a graph that contains no path consisting of four vertices as a vertex-induced subgraph?

Cograph Modification is a particularly interesting problem because, when restricted to cographs, many NP-hard problems are solvable in polynomial time [8].

Results. We based our algorithm on the characterization of cographs as graphs containing no induced path of four vertices (P_4). We use an expansion rule analogous to the one used for Triangle Vertex Deletion: we assume every vertex being part of a P_4 is part of at least two P_4's. We get a worst-case branching number of 3.30 when examining graphs up to seven vertices. This matches the bound given for 4-Hitting Set in [31].

Table 6: Summary of search tree sizes for the problems considered

Problem	Trivial	Best known result	Our method
Cluster Editing	3	2.27 [18]	1.92
Cluster Deletion	2	1.77 [18]	1.53
Cluster Vertex Deletion	3	2.27 [31]	2.26
Triangle Edge Deletion	3	2.27 [31]	2.47
Triangle Vertex Deletion	3	2.27 [31]	2.42
Cograph Vertex Deletion	4	3.30 [31]	3.30

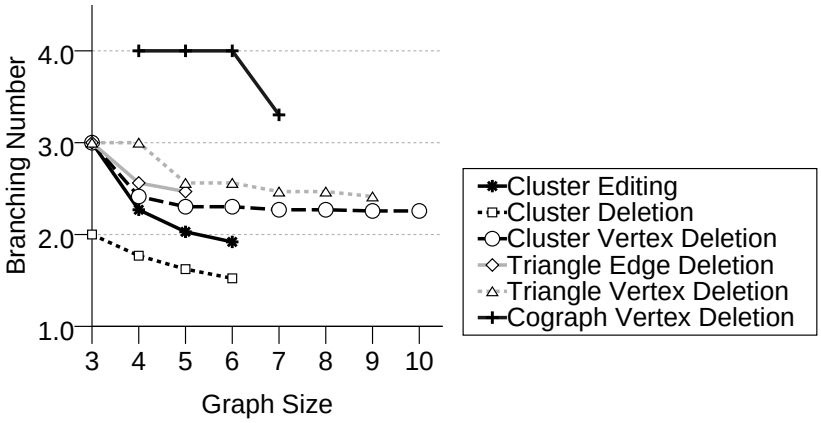

Figure 6: Worst-case branching number depending on size of considered subgraphs

4.3 Summary

Focusing on the worst-case branching numbers computed for various graph modification problems, we give an overview on our results in Table 6: We compare the worst-case branching numbers corresponding to a trivial branching, the best so far known result, and the search tree algorithm size bound computed by our method.

In Fig. 6, 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 an improved worst-case branching number.

We summarize some further observations as follows:

- In many cases, the average branching number of the computed branch-

ing rules is significantly smaller than the worst-case one.

- For smaller graphs, a larger part of the running time is spent on the isomorphism tests. With growing graph sizes, the part of the running time spent on the administration of branching vectors in the search tree becomes larger and often takes close to 100 percent of the running time.

- The resulting branching rules branch, even for large graphs, only into a moderate number of branching cases, e.g., into at most 11 branching cases in Cluster Vertex Deletion when inspecting graphs of size 10.

5 Conclusion

We presented a software tool to automatically generate search tree algorithms for graph modification problems. Further details on the whole scenario and the implementation can be found in [21].

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 problems. 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, often highly complicated case distinctions. In this sense, our framework helps to reveal the usually few real "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, it seems feasible to apply the basic framework not only to graph modification problems (see [21] for more considerations in this direction).

Open problems and challenges. Many things remain to be done. Obviously, trying to develop new problem-specific rules may lead to improved search tree size bounds. Further graph modification problems could be considered. It remains future work 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: the enumeration of all non-isomorphic graphs up to a certain size and the concatenation of (large sets of) branching vectors in our meta search tree. It is a subject of ongoing work to design improved (maybe also heuristic) strategies for these tasks. In this way, the obtained upper bounds on search tree sizes might also be further improved. 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; for instance, we recently succeeded in finding a non-trivial bound for the NP-hard Dominating Set problem with a maximum vertex degree of 3 [21]. Here, we so far obtained the search tree size $O(3.71^k)$, where k denotes the number of vertices in the dominating set. The trivial bound is $O(4^k)$. Finally, a challenge could also be to use the automated framework in order to derive analytical proofs for search tree sizes, i.e., proofs and case distinctions that can again be verified by hand.

Related work. Independently from this work, Frank Kammer and Torben Hagerup (Augsburg) informed us about ongoing related work concerning computer-generated proofs for upper bounds on hard combinatorial problems, such as Independent Set and Maximum Satisfiability. In another independent piece of work, Nikolenko and Sirotkin [33] describe how to automate proofs for worst-case upper bounds based on search trees for the Satisfiability problem. Their approach, however, seems more problem-specific and more ad hoc than ours and does not provide a general automation framework. Similar studies have also been undertaken by Fedin and Kulikov [14] in studying a special variant of the Maximum Satisfiability problem. Eventually, we note that Robson [35] also used the computer in order to improve the search tree of his algorithm for Independent Set [34]. 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.

A Sample Program Output

In the following, we will present the output of our program which describes the algorithm from Theorem 2, which solves Cluster Editing in $O(1.92^k +$

$|V|^3$) time.

In the program output, graphs are given with *GraphIDs*. A GraphID is an integer which compactly encodes undirected graphs (where vertices are numbered by $0, 1, 2, \ldots$) with a binary representation: 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 $\sum_{i=0}^{v-1} i + u$. Of all possible vertex numberings for a graph, we choose the one which yields the smallest GraphID. For example, the P_3 contains the edges $\{0, 1\}$ and $\{0, 2\}$, encoded in bits 0 and 1, yielding the GraphID $2^0 + 2^1 = 3$.

As elaborated in Sect. 4.1.1, the algorithm first locates an induced P_3 in the input and successively adds neighboring vertices until a size-6 subgraph is generated by this expansion process. It then branches into several cases depending on the expanded subgraph.

The first part of the output describes the expansion phase. Each line (wrapped here for reading convenience) describes how to specifically expand certain subgraphs:

```
  3 1.920 edge-expand {0, 1} 15 31
 15 1.920 edge-expand {0, 3} 235 239 255
 31 1.873 add-vertex 95 119 223 239 254 255 507 511
 95 1.840 edge-expand {0, 4} 3307 3311 3451 3455 3579 3583
119 1.863 edge-expand {0, 4} 3879 3887 3903 3949 3959 3967
223 1.859 add-vertex 1247 1463 3295 3311 3327 3577 3579 3583
                     7673 7675 7679
235 1.920 add-vertex 1259 1269 3307 3443 3451 3879 3947 3949
                     3951 7902 7903
239 1.873 add-vertex 1263 1271 1469 3311 3447 3451 3453 3455
                     3579 3887 3951 3958 3959 3967 7915 7917
                     7919 7934 7935
254 1.804 edge-expand {0, 4} 3949 7902 7917 7934 8185 8187
255 1.827 add-vertex 1279 1471 1975 3327 3455 3583 3903 3951
                     3966 3967 4095 7903 7919 7935 8185 8187
                     8191
507 1.824 add-vertex 1531 1533 3579 3582 3959 7675 7919 7934
                     8187 16350 16351
511 1.794 add-vertex 1535 1983 3583 3967 4094 7679 7935 8187
                     8191 16351 16383
```

The first column denotes the GraphID of the subgraph. The second column is the worst-case branching number we will obtain when encountering this subgraph. The third column tells what to do with this subgraph: if

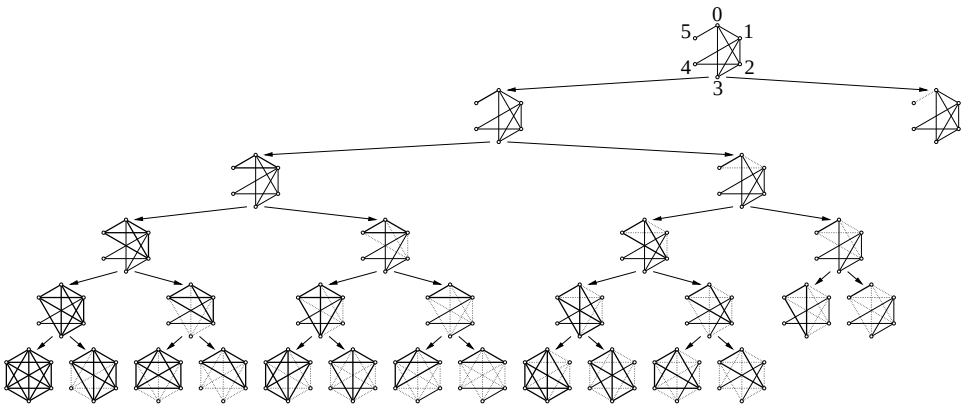

Figure 7: Branching for graph 1471. *Bold lines* denote permanent edges, *dotted lines* denote forbidden edges

it is `edge-expand {`x`,` y`}`, then one has to find the common neighbor of x and y and add it to the subgraph. If it is `add-vertex`, then one has to add any vertex adjacent to a vertex of the subgraph. The remaining part of each line lists the graphs which can result from the expansion.

The expansion of a P_3 leads to one of 57 different size-6 graphs. For each of them, a branching rule is calculated. We give only one example because of space constraints, namely for graph 1471, which is generated from the P_3 via graphs 15 and 255. The branching contains 15 cases and has the branching vector $(6, 5, 7, 7, 9, 6, 8, 6, 9, 6, 8, 6, 5, 3, 1)$, corresponding to the branching number 1.826. This branching rule is also sketched in Fig. 7. The program output for this graph, wrapped and indented, looks like this:

```
({0, 5}:
     ({0, 1}:
          ({0, 2}:
               ({0, 3}:
                    ({0, 4}: *, *),
                    ({0, 4}: *, *)),
               ({0, 3}:
                    ({0, 4}: *, *),
                    ({0, 4}: *, *))),
          ({0, 2}:
               ({0, 3}:
                    ({0, 4}: *, *),
```

```
                ({0, 4}: *, *)),
          ({0, 3}: *, *))),
*)
```

Each node of the tree is described by a term `({`x`, `y`}: `p`, `f`)`, where $\{x, y\}$ denotes the vertex pair to branch on, p is the branching case where the vertex pair is marked permanent, and f is the case where the vertex pair is marked forbidden. Leaves are marked with `*`. As one can see in Fig. 7, frequently edges are additionally marked permanent or forbidden because of the application of the reduction rules; this is not represented explicitly in the textual form.

The output with the complete list of branching rules can be downloaded from http://www-fs.informatik.uni-tuebingen.de/piaf/software/ce-1.92.

References

[1] J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229:3–27, 2001.

[2] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. In *Proc. 43rd FOCS*, pages 238–247. IEEE Computer Society, 2002.

[3] N. Bansal and V. Raman. Upper bounds for MaxSat: further improved. In *Proc. 10th ISAAC*, volume 1741 of *LNCS*, pages 247–258. Springer, 1999.

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

[5] M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. In *Proc. 44rd FOCS*, pages 524–533. IEEE Computer Society, 2003.

[6] J. Chen and I. A. Kanj. Improved exact algorithms for MAX-SAT. In *Proc. 5th LATIN*, volume 2286 of *LNCS*, pages 341–355. Springer, 2002.

[7] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[8] D. G. Corneil, Y. Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[9] V. Dahlhöf and P. Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proc. 13th ACM-SIAM SODA*, pages 292–298, 2002.

[10] E. Dantsin, E. A. Hirsch, S. Ivanov, and M. Vsemirnov. Algorithms for SAT and upper bounds on their complexity. Technical Report TR01-012, Electronic Colloquium on Computational Complexity, 2001.

[11] E. D. Demaine and N. Immorlica. Correlation clustering with partial information. In *Proc. 6th APPROX*, volume 2764 of *LNCS*. Springer, 2003.

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

[13] D. Emanuel and A. Fiat. Correlation clustering — minimizing disagreements on arbitrary weighted graphs. In *Proc. 11th ESA*, volume 2832 of *LNCS*, pages 208–220. Springer, 2003.

[14] S. S. Fedin and A. S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. Manuscript, Steklov Institute of Mathematics, St. Petersburg, September 2003.

[15] M. R. Fellows. Parameterized complexity: The main ideas and connections to practical computing. In *Experimental Algorithmics*, volume 2547 of *LNCS*, pages 51–77. Springer, 2002.

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

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

[18] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: fixed-parameter algorithms for clique generation. In *Proc. 5th CIAC*, volume 2653 of *LNCS*, pages 108–119. Springer, 2003. Long version to appear in *Theory of Computing Systems*.

[19] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. *Discrete Applied Mathematics*, 130(2):139–155, 2003.

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

[21] F. Hüffner. *Graph Modification Problems and Automated Search Tree Generation*. Diploma thesis, Universität Tübingen, WSI für Informatik, October 2003.

[22] M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, 1986.

[23] A. S. Kulikov. An upper bound $O(2^{0.16254n})$ for Exact 3-Satisfiability: A simpler proof. *Zapiski nauchnyh seminarov POMI*, 293:118–128, 2002. English translation is to appear in *Journal of Mathematical Sciences*.

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

[25] X. Leroy, J. Vouillon, D. Doligez, et al. The Objective Caml system. Software and documentation available on the web, `http://caml.inria.fr/`, 1996.

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

[27] M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.

[28] B. D. McKay. **nauty** user's guide (version 1.5). Technical report TR-CS-90-02, Australian National University, Department of Computer Science, 1990.

[29] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001.

[30] R. Niedermeier and P. Rossmanith. New upper bounds for Maximum Satisfiability. *Journal of Algorithms*, 36:63–88, 2000.

[31] R. Niedermeier and P. Rossmanith. An efficient fixed parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 1:89–102, 2003.

[32] R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for Weighted Vertex Cover. *Journal of Algorithms*, 47(2):63–77, 2003.

[33] S. I. Nikolenko and A. V. Sirotkin. Worst-case upper bounds for SAT: automated proof. Presented at *15th European Summer School in Logic Language and Information (ESSLLI 2003)*, 2003.

[34] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.

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

[36] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. In *Proc. 28th WG*, volume 2573 of *LNCS*, pages 379–390. Springer, 2002.

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

[38] R. Sharan and R. Shamir. CLICK: A clustering algorithm with applications to gene expression analysis. In *Proc. 8th ISMB*, pages 307–316. AAAI Press, 2000.

[39] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Proc. 5th International Workshop on Combinatorial Optimization*, volume 2570 of *LNCS*, pages 185–208. Springer, 2003.