PRIFYSGOL CYMRU ABERTAWE
UNIVERSITY OF WALES SWANSEA

UNIVERSITY OF WALES SWANSEA

REPORT SERIES

Decomposing clause-sets: Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNF's

by

Marijn Heule and Oliver Kullmann

Report # CSR 2-2006

Decomposing clause-sets: Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNF's

Marijn Heule*
Department of Software Technology
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: marijn@heule
http://www.st.ewi.tudelft.nl/~heule/

Oliver Kullmann†
Computer Science Department
University of Wales Swansea
Swansea, SA2 8PP, UK
e-mail: O.Kullmann@Swansea.ac.uk
http://cs-svr1.swan.ac.uk/~csoliver/

March 19, 2006

Contents

*Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.306
†Supported by EPSRC Grant GR/S58393/01

Abstract

Applications of graph and hypergraph decomposition techniques to SAT solving have been studied under various conditions. Theoretical research focused on the direction given by *tree decompositions*, while practical approaches either integrated tree decomposition guidance into existing SAT solver, or studied splitting techniques using *hypergraph cuts*. In this article we present a uniform and general framework based on the splitting approach. We allow (and study) different graph representations, different graph and hypergraph splitting methods (involving vertices or (hyper)edges), different degrees of integration with the (full) problem structure, and various heuristics for selecting "good" decompositions. Basic is the notion of a *decomposable graph representation*. We show that *resolution graphs* (with only non-tautological resolvents(!)) yield a decomposable graph representation. Tree decomposition and variations have theoretical strength, but for practical purposes the (hyper)graph cut approach seems to be more suited, which is of a local nature, yielding one split at a time, while tree decomposition delivers one global scheme, from which all future splittings are derived. For (hyper)graph cuts even the weaker variation of resolution graphs, the *conflict graphs* (where also tautological resolvents are allowed), yield strictly better splitting possibilities than variable interaction graphs (at the expense of using bigger graphs). Very preliminary experimental results are given (yet mostly of negative nature).

1 Introduction

1.1 SAT decision via graph/hypergraph decomposition in general

Given a "SAT problem" F (which could be a boolean CNF, or a constraint satisfaction problem), the basic approach is to use some *graph representation* $G(F)$ (which actually might be a graph or a hypergraph) representing certain elements of F. Typically this representation is not faithful, but $G(F)$ is only a rough representation of F — which is a weakness as well as a strength. In this article we only consider exploitation of *connectedness properties* of $G(F)$ (of course many other applications exist) in the following way: If $G(F)$ splits into connected components $G_1, \ldots, G_m$, then the representation should allow us to efficiently find a partitioning $F_1, \ldots, F_m$ of F with $G(F_i) = G_i$ for $i \in \{1, \ldots, m\}$, such that

$$\mathrm{sat}(F) = \min_{i \in \{1,\ldots,m\}} \mathrm{sat}(F_i), \tag{1}$$

where $\mathrm{sat}(F) = 1$ if F is satisfiable, while $\mathrm{sat}(F) = 0$ otherwise. In other words, if one of the "components" F_i of F is unsatisfiable, then F is unsatisfiable, while if all components F_i are satisfiable, then F is satisfiable. Property (1) is the basic requirement on a useful graph representation (in this article we will give an example, where establishing (1) is not obvious).

We can apply (1) "opportunistically" in a SAT solver, by either only inspecting the input, or also investigating instances in the backtracking tree. Going one step further, one can instrument the SAT solver somehow, so that it might produce decompositions by itself, or that it might implicitly exploit (1), as done in [4].[1)] In this article we will not investigate this kind of connection between (hyper)graph decomposition and SAT algorithm, since this connection seems to depend strongly on the specific form of graph representation *and* on the specific method of decompositions employed. One goal of this article is to create a general framework for arbitrary graph representations and arbitrary methods of splitting graphs, in this way also unifying approaches based on hypergraph splitting (as in [11]) and tree decomposition (as in [35]).

So for us the basic questions are:

1. How can we split (hyper)graphs at the graph side?
2. How can we realise this on the formula side?

Let G be a connected graph or a connected hypergraph. If G is a graph, then a *separating vertex set* is a set of vertices such that after removing them (including incident edges) the graph splits into components, while a *separating edge set* is a set of edges such that removing them the graph splits into components. For hypergraphs G one usually does not consider separating vertex sets, since they can be realised as separating edge sets in the dual hypergraph G^t. A *separating hyperedge set* for a hypergraph is a set of (hyper)edges such that removing them splits the hypergraph. So for graphs we have separating vertex sets and separating edge sets, while for hypergraphs we have separating hyperedge sets generalising separating edge sets for graphs. In principle we do not need to consider hypergraphs,

[1)] See below for an explanation, how tree decomposition fits into our general scheme.

but we can reduce all these notions to the notion of separating vertex sets in graphs as follows: Separating hyperedge sets in a hypergraph G can be (nearly perfectly) mirrored by separating vertex sets in the line graph of G, while separating hyperedge sets in a the dual G^t can be (nearly perfectly) mirrored by separating vertex sets in the 2-section of G. So for the more principle considerations in this paper we will often just refer to the graph case, but we want to point out here that for efficiency of algorithms and implementations as well as for clarity of theoretical insight often the hypergraph approach is superior.

A remark on "cuts" versus "separating sets": We are using "separating sets" ("disconnecting sets"), since while cuts do cut "something", we just want to cut "anything", and because of that the minimality condition included in the notion of a cut is not completely appropriate for us (see [33] for these notions). However in this introduction we ignore these differences, and freely switch between "cuts" and "separating (vertex/edge/hyperedge) sets".

Now how to realise such separating sets on the formula side? In the cases we consider, vertices / edges correspond to clauses or variables: Removing a variable happens by assigning a truth value to it, while for removing a clause we need to find a partial assignment satisfying the clause. If we have a set of k variables, then we consider all 2^k assignments, while if we have a set F' of clauses then we consider a set of satisfying partial assignments covering all satisfying assignments (this is exactly the same as transforming the CNF F' into a DNF — the clauses of the DNF just correspond to the satisfying partial assignments). In both cases, we can apply further reductions like unit-clause-propagation.

1.2 General algorithms based on graph splitting

In Section 3 we present a generic scheme for the exploitation of graph splittings (in a very general context, not restricted to SAT), resulting in the generic algorithm $\mathcal{GS}$, which is the basis for all the algorithms we study here. We will study several levels of refinements for this scheme. One basic problem of this approach is to find a "good split", and in Section 4 we present the refinement $\mathcal{HS}$, which shows a natural (and not completely trivial) approach for selecting a "best split" from a given set of possibilities.

Regarding tree decompositions, we take a slightly unusual point of view (a selection of the literature relevant for SAT is given by [31],[15],[1],[35],[6],[4],[30]): We use a tree decomposition of a graph as a "universal splitting advice", and choosing central splitting points in the tree decomposition itself we arrive at a "splitting tree" for the graph, which contains a complete splitting scheme of the graph into "trivial parts". The algorithm $\mathcal{TS}$ obtained in this way is given in Subsection 5.3. We see that in this way (as a special instance of $\mathcal{GS}$) we do not achieve fixed parameter tractability (FPT), but only polynomial time problem solving in the treewidth, but our method is applicable under very general conditions, and can be combined with local reasoning (like unit clause propagation as mentioned above).

After having set the stage with the three generic algorithms $\mathcal{GS}$, $\mathcal{HS}$ and $\mathcal{TS}$, in Section 6 we turn to the question what kind of graph representations to use for solving SAT problems.

1.3 The graph representations considered in this paper

Given a clause-set F, it is natural to consider the *variable hypergraph* of F, whose vertices are the variables of F, and whose hyperedges are the sets of variables given by the clauses of F. For this hypergraph tree decomposition has been (quite intensively) studied, as well as hypergraph cuts for the dual hypergraph. For practical purposes the hypergraph representation is superior over the graph representation (since it's more faithful *and* more efficient), however for theoretical purposes it is more convenient to consider vertex cuts of the *variable intersection graph* or edge cuts for the *common variable graph*, where the first graph representation is variable-based, while the second is clause-based. Ignoring implementation aspects, all these approaches have exactly the same power. Tree decomposition (since based on vertex cuts) employs the variable interaction graph, while hypergraph cuts employ the common variable graph (for theoretical purposes).

Considering the common variable graph, we see that it has not much "decomposition intelligence" (and so neither has the variable interaction graph). Strictly more powerful is the *conflict graph*(w.r.t. edge cuts), where two vertices (clauses) are joined iff they clash in at least one literal. It is not too hard to show that also here all properties of a decomposable graph representation are fulfilled. Less trivial is that the still strictly stronger concepts of *resolution graph* and *resolution modulo subsumption graph* (where two clauses are joined iff they clash in *exactly* one literal resp. if they clash in exactly one literal and the resolvent is not subsumed by an already existing clause) yield a decomposable graph representation; the only existing proof seems to be in [20], and we will discuss this proof here.

For the common variable graph, the conflict graph and the resolution graphs we have two splitting possibilities: Splitting on vertices (used in tree decompositions), and splitting on edges. Splitting on vertices seems out of question for the common variable graph, but becomes a more interesting option for the resolution graphs. Especially interesting seem edge cuts, where edge cuts for the common variable graph "noisily correspond" to the standard decomposition approach for the variable interaction graph or the (dual) variable hypergraph, while with edge cuts for the other graph types we gain more decomposition power (at the cost of larger graphs).

1.4 Towards efficient SAT algorithms

In Subsection 7.1 we introduce the concept of a *decomposable graph representation*, which specialises (and makes more precise) the general framework of graph decomposition for the case of SAT decision, where now separating vertex or (hyper)edge sets are "realised" by the application of partial assignments. The problem of how to find these partial assignments is discussed in Subsection 7.2.

Having general schemes for applying graph splitting as well as several graph representations of clause-sets at hand, and knowing several possibilities of how to realise the schemes for SAT decision, we are ready to turn these general approaches into practice. Unfortunately, at this time we do not have data on running any of the proposed algorithms, but we have only data on the sizes of the various graph representation, and how approximation algorithms for the treewidth behave on them (see Section 8).

Some final remarks on the literature (we are aware of) on decomposition applications specifically for (general) SAT solving:

1. In [3] the variable interaction graph is considered, and the point is to observe (and react) to decompositions, but not to actively enforcing decompositions.

2. Closer to our approach is [11], which uses hypergraph decomposition; again only the variable interaction graph is used (in its (dual) hypergraph version).

2 Preliminaries

2.1 Labelled clause-sets

Clauses are finite and complement-free (non-tautological) sets of literals, the set of all clauses is $\mathcal{CL}$. Complementation of literals x is written as $\overline{x}$, while complementation of clauses is applied elementwise. The underlying variable of a literal x is denoted by $\mathrm{var}(x)$, while for a clause C we set $\mathrm{var}(C) := \{\mathrm{var}(x) : x \in C\}$. Two clauses C, D are resolvable if they clash in exactly one literal, that is, if $|C \cap \overline{D}| = 1$ holds. If C, D are resolvable, then their resolvent is denoted by $C \diamond D := (C \setminus \{x\}) \cup (D \setminus \{\overline{x}\})$ for $C \cap \overline{D} = \{x\}$, where x is called the resolution literal (occurring positively in the left parent clause, negatively in the right parent clause). Two resolvable clauses C, D are resolvable on variable v if for the resolution literal x we have $\mathrm{var}(x) = v$.

We use standard notations for multi-clause-sets as in [24], but we promote multi-clause-sets (where we can have multiple occurrences of clauses, but these multiple occurrences are not distinguishable) to *labelled clause-sets*, which are triples (V, L, F) where V is the set of variables, L is the set of clause labels, and $F : L \to \mathcal{CL}$ is the clause-function such that for all $l \in L$ we have $\mathrm{var}(F(l)) \subseteq V$. So labelled clause-sets are very similar to general hypergraphs, only that instead of hyperedges (sets of vertices) we use clauses (sets of variables with polarities). As in the case of general hypergraphs we have the notions $F \subseteq F'$ and $\bigcup_{i \in I} F_i$ as well as $\bigcup\!\!\!\!\cdot_{i \in I} F_i$ for labelled clause-sets F, F' and F_i, and a (finite) set of clauses F gets promoted to a labelled clause-sets by using the clauses as clause-labels. We typically identify a labelled clause-sets (V, L, F) with its clause-function F, and we write $\mathrm{var}(F) = V$ for the set of variables; if we need to name the clause-labels, then we use $L(F)$.

The application of a partial assignment φ to a labelled clause-set F is denoted by $\varphi * F$, and is defined as the following labelled clause-set:

1. $\mathrm{var}(\varphi * F) := \mathrm{var}(F) \setminus \mathrm{var}(\varphi)$;

2. $L(\varphi * F)$ is the set of $l \in L(F)$ such that the clause $F(l)$ is not satisfied by φ;

3. for $l \in L(\varphi * F)$ let $(\varphi * F)(l)$ be the clause obtained from $F(l)$ by removing all literals falsified by φ.

The number of variables in F is denoted by $n(F) := |\mathrm{var}(F)|$, the number of clauses in F by $c(F) := |L(F)|$ (thus from $F = \bigcup\!\!\!\!\cdot_{i \in I} F_i$ it follows $c(F) = \sum_{i \in I} c(F_i)$).

The empty clause is denoted by $\bot$, the empty labelled clause-set by $\top$ (no variables, no clause-labels). Thus a labelled clause-set F is satisfiable iff there exists a partial assignment φ with $\varphi * F = \top$, and φ is then called a satisfying partial assignment for F. By $\mathrm{mod_t}(F)$ we denote the set of partial assignments φ with $\varphi * F = \top$ and $\mathrm{var}(\varphi) = \mathrm{var}(F)$.

The composition of partial assignments φ, ψ is denoted by $\varphi \circ \psi$ ("first ψ, then φ"). If Φ, Ψ are sets of partial assignments, then we use the usual notation $\Phi \circ \Psi :=$

$\{\varphi \circ \psi : \varphi \in \Phi \wedge \psi \in \Psi\}$ for the complex product. If $(\Phi_1, \ldots, \Phi_n)$, $n \in \mathbb{N}_0$, is a finite sequence of sets of partial assignments, then $\circ_{i=1}^n \Phi_i$ is well-defined (since the partial assignments together with composition form a monoid). Two partial assignments φ, ψ commute ($\varphi \circ \psi = \psi \circ \varphi$) iff they are compatible (common variables get assigned the same value). So if $(\Phi_i)_{i \in I}$ is a finite family of partial assignments such that for $i, j \in I$, $i \neq j$ all $\varphi \in \Phi_i$ commute with all $\psi \in \Phi_j$, then also $\circ_{i \in I} \Phi_i$ is defined.

2.2 Graphs

A *finite graph* G is a pair $G = (V, E)$ where $V = V(G)$ is the finite vertex set and $E = E(G) \subseteq \binom{V}{2} = \{\{a, b\} : a, b \in V \wedge a \neq b\}$ is the edge set. Thus graphs do not contain loops nor multiple edges (we are interested only in connectivity issues are, and thus, different from [24, 9] multiple edges do not play a role here). In this report all graphs considered are finite, and so we leave out the adjective "finite" for graphs in the sequel. The *null graph* is the graph with no vertices, while we do not use a special notion for graphs without edges (obviously the null graph has no edges). To avoid formal effort we assume that always $V(G) \cap E(G) = \emptyset$ holds "automagically".

In a "general graph" where parallel vertices (and loops) are allowed, a "walk" is an alternating sequence of vertices and edges; since we do not need general graphs here, we only use the simpler version of walks as sequences of vertices (which suffices since in graphs edges are either present or absent, but they do not have "personalities"). For vertices $v, w \in V(G)$ in a graph G a *walk* from v to w in G is a sequence $v_0, v_1, \ldots, v_n$ of vertices $v_i \in V(G)$ for $n \in \mathbb{N}_0$ with $v_0 = v$, $v_n = w$ such that for all $i \in \{0, \ldots, n-1\}$ the vertices v_i, v_{i+1} are adjacent in G (i.e., $\{v_i, v_{i+1}\} \in E(G)$). The length of the walk is n, while the walk is a *path* if for $i, j \in \{0, \ldots, n\}$, $i < j$ in case of $v_i = v_j$ we have $i = 0$ and $j = n$; furthermore the walk is *closed* if $v_0 = v_n$. Finally the walk is called a *circuit* if it is closed path of length at least 1.

Two vertices $v, v' \in V(G)$ are *connected* in G if a walk from v to v' in G exists (which is equivalent to the existence of a path from v to v'). A graph G is *connected* if all vertices $v, v' \in V(G)$ are connected. A graph G is called *acyclic* if no circuit exists in G.

A *tree* T is a connected acyclic graph with non-empty vertex set, while a *forest* is (simply) an acyclic graph. A graph G is a forest if and only if for all connected vertices $v, v' \in V(G)$ there exists exactly one path from v to v' in G.

If G is a graph and $M \subseteq V(G)$, then $\boldsymbol{G - M}$ is the graph with vertex set $V(G) \setminus M$ and edge set $\{e \in E(G) : e \cap M = \emptyset\}$. On the other hand, if $N \subseteq E(G)$, then $\boldsymbol{G \setminus N}$ is the graph with vertex set $V(G)$ and edge set $E(G) \setminus N$.

A graph G' is called a *subgraph* of G (written as $G' \subseteq G$) if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$ holds, while on the one hand a subgraph G' is *induced* if there is $M \subseteq V(G)$ with $G' = G - M$, and on the other hand a *partial graph* of G is a subgraph G' such that there is $N \subseteq E(G)$ with $G' = G \setminus N$.

A *connected component* of G is a maximal connected subgraph (thus necessarily induced) with non-empty vertex set, and the set of all connected components of G is denoted by $\mathbf{cc}(\boldsymbol{G})$.

If $(G_i)_{i \in I}$ is a family of graphs, then by $\bigcup_{\boldsymbol{i \in I}} \boldsymbol{G_i}$ we denote the graph with vertex set $\bigcup_{i \in I} V(G_i)$ and edge set $\bigcup_{i \in I} E(G_i)$; in case the family $(V(G_i))_{i \in I}$ of vertex

sets is disjoint we may write $\bigcupdot_{i\in I} G_i$. Thus for every graph G we have $G = \bigcupdot_{G'\in \mathrm{cc}(G)} G'$, and $\mathrm{cc}(G)$ is the unique set S of non-null subgraphs of G with $G = \bigcupdot_{G'\in S} G'$. A graph G is connected if and only if from $G = G_1 \cupdot G_2$ it follows that at least one of G_1 or G_2 is the null graph.

IF G is connected, then a *separating vertex set* of G is a set $M \subseteq V(G)$ such that $G - M$ is not connected (disconnected), while a *separating edge set* of G is a set $N \subseteq E(G)$ such that $G \setminus N$ is not connected.

2.3 Graphs with bipartition (incidence structures)

A *bipartition* of a graph G is a pair (A, B) with

1. $A \cap B = \emptyset$
2. $A \cup B = V(G)$
3. $E(G) \subseteq \{\{a, b\} : a \in A \wedge b \in B\}$.

A *graph with bipartition* (or *incidence structure*) is a pair $P = (G, (A, B))$ such that G is a graph and (A, B) is a bipartition of G. We write $A = A(P)$ (the "first part"), $B = B(P)$ (the "second part") and $G = U(P)$ (the "underlying graph"). A graph with bipartition can be implicitly converted to its underlying graph. The *null graph with bipartition* is the pair $((\emptyset, \emptyset), (\emptyset, \emptyset))$.

If $P = (G, (A, B))$ is a graph with bipartition, then the *transposition* $\boldsymbol{P^t}$ is $(G, (B, A))$ (which again is a graph with bipartition). Obviously we have $(P^t)^t = P$.

For a family $(P_i)_{i\in I}$ of graphs with bipartition we write $\bigcup_{i\in I} P_i$ for the graph with bipartition with underlying graph $\bigcup_{i\in I} U(P_i)$, first part $\bigcup_{i\in I} A(P_i)$ and second part $\bigcup_{i\in I} B(P_i)$. We may write $\bigcupdot_{i\in I} P_i$ if we can write $\bigcupdot_{i\in I} U(P_i)$, $\bigcupdot_{i\in I} A(P_i)$ and $\bigcupdot_{i\in I} B(P_i)$.

A graph with bipartition G is connected if from $G = G_1 \cupdot G_2$ it follows that at least one of G_1, G_2 is the null graph with bipartition. Thus G is connected iff the underlying graph is connected.

For a graph with bipartition $P = (G, (A, B))$ we define the *bridging graphs* $\boldsymbol{\mathrm{Br}_i(P)}$, $i \in \{1, 2\}$, as the graph with vertex set A resp. B, such that two vertices are joined by an edge if their distance in G is (exactly) 2. Obviously $\mathrm{Br}_i(P) = \mathrm{Br}_{3-i}(P^t)$.

If P has no isolated vertices in $B(P)$, then P is connected iff $\mathrm{Br}_1(P)$ is connected. And if P has no isolated vertices in $A(P)$, then P is connected iff $\mathrm{Br}_2(P)$ is connected.

2.4 Hypergraphs

For literature see [2, 7] and Chapters 1, 2 in [16].

A *finite general hypergraph* is a triple $G = (V, E, \mathcal{E})$, where $V = V(G)$ is the finite vertex set, $E = E(G)$ is the finite set of labels for hyperedges, and $\mathcal{E} = \mathcal{E}(G) : E \to \mathbb{P}(V) = \{T : T \subseteq V\}$ assigns to every hyperedge label a subset of V. In this report all general hypergraphs considered are finite, and so we leave

out the adjective "finite" for hypergraphs in the sequel. To avoid formal effort we furthermore always assume $V(G) \cap E(G) = \emptyset$ "automagically". The *(general) null hypergraph* is the general hypergraph with empty vertex set and empty edge label set. A general hypergraph G contains an empty hyperedge if there exists $e \in E(G)$ with $\mathcal{E}(G)(e) = \emptyset$, while G contains a singleton hyperedge if there is $e \in E(G)$ with $|\mathcal{E}(G)(E)| = 1$. A vertex $v \in V(G)$ is *non-covered* if $v \notin \bigcup_{e \in E(G)} \mathcal{E}(G)(e)$.

A *hypergraph* G is a pair (V, E) where $V = V(G)$ is the finite vertex set and $E = E(G) \subseteq \mathbb{P}(V) = \{T : T \subseteq V\}$ is the set of hyperedges; a hypergraph G can be implicitly converted to the general hypergraph with vertex set $V(G)$, hyperedge set $E(G)$ and hyperedge function the identity. Every graph is a hypergraph.

The *dual* $\boldsymbol{G^t}$ of a general hypergraph G ("t" like "transposition") is given by

1. $V(G^t) = E(G)$
2. $E(G^t) = V(G)$
3. $\mathcal{E}(G^t) = (e \in E(G^t) \mapsto \{v \in V(G^t) : e \in \mathcal{E}(G)(v)\})$.

Obviously we have $(G^t)^t = G$. G contains a non-covered vertex iff G^t contains an empty edge, and G contains an empty edge iff G^t contains a non-covered vertex.

The *vertex-hyperedge-graph* $\mathbf{bip(G)}$ (or the "associated bipartite graph", if there is no danger of confusion) of a general hypergraph G is the graph with bipartition $(U, (V(G), E(G)))$, where U is the graph with vertex set $V(G) \dot\cup E(G)$ and edge set $\{\{v, E\} : E \in E(G), v \in e\}$. The *hyperedge-vertex-graph* of G is $\mathrm{bip}(G^t) = \mathrm{bip}(G)^t$.

Given a graph with bipartition G, the associated general hypergraph $\mathbf{hyp(G)}$ is the general hypergraph with vertex set $A(G)$, hyperedge label set $B(G)$ and hyperedge function which maps every $E \in B(G)$ to the set of elements of $A(G)$ which are adjacent to E in $U(G)$. We have $\mathrm{hyp}(\mathrm{bip}(G)) = G$ for every general hypergraph G, and $\mathrm{bip}(\mathrm{hyp}(G)) = G$ for every graph with bipartition G. Thus $\mathrm{hyp}(G^t) = \mathrm{hyp}(G)^t$.

For two general hypergraphs G_1, G_2 where $\mathcal{E}(G_1)$ and $\mathcal{E}(G_2)$ are compatible (that is, for $e \in E(G_1) \cap E(G_2)$ we have $\mathcal{E}(G_1)(e) = \mathcal{E}(G_2)(e)$) we define $\boldsymbol{G_1 \cup G_2}$ as the general hypergraph given by

$$\begin{aligned} V(G_1 \cup G_2) &= V(G_1) \cup V(G_2) \\ E(G_1 \cup G_2) &= E(G_1) \cup E(G_2) \\ \mathcal{E}(G_1 \cup G_2) &= \mathcal{E}(G_1) \cup \mathcal{E}(G_2). \end{aligned}$$

We may write $G_1 \dot\cup G_2$ in case we have $V(G_1) \cap V(G_2) = \emptyset$ and $E(G_1) \cap E(G_2) = \emptyset$. Similarly for arbitrary families $(G_i)_{i \in I}$ of general hypergraphs. We have $\mathrm{bip}(\bigcup_{i \in I} G_i) = \bigcup_{i \in I} \mathrm{bip}(G_i)$ and $\mathrm{bip}(\bigdot\bigcup_{i \in I} G_i) = \bigdot\bigcup_{i \in I} \mathrm{bip}(G_i)$, and for families $(G_i)_{i \in I}$ of graphs with bipartition we have $\mathrm{hyp}(\bigcup_{i \in I} G_i) = \bigcup_{i \in I} \mathrm{hyp}(G_i)$ and $\mathrm{hyp}(\bigdot\bigcup_{i \in I} G_i) = \bigdot\bigcup_{i \in I} \mathrm{hyp}(G_i)$

For a general hypergraph G and $X \subseteq V(G) \dot\cup E(G)$ (recall our general assumption) we write $\boldsymbol{G \setminus X}$ for the general hypergraph with vertex set $V(G) \setminus X$, hyperedge label set $E(G) \setminus X$ and hyperedge function the obvious restriction (for domain and images) of $\mathcal{E}(G)$. Thus $\mathrm{bip}(G \setminus X) = \mathrm{bip}(G) - X$.

We use $\boldsymbol{G^*} := G \setminus \{e \in E(G) : \mathcal{E}(G)(e) = \emptyset\}$ (removing all empty edges) and $\boldsymbol{G_*} := ((G^t)^*)^t$ (removing all non-covered vertices). Thus a general hypergraph G has no empty edges iff $G^* = G$ holds, while G has no non-covered vertex iff $G_* = G$ holds.

For a general hypergraph G we define the 2-section $\boldsymbol{G_{[2]}}$ as the graph with vertex set $V(G)$ and edge set $\bigcup_{e \in E(G)} \binom{\mathcal{E}(G)(e)}{2}$ (that is, two vertices are adjacent iff there is some hyperedge containing both of them). Thus $G_{[2]} = \mathrm{Br}_1(\mathrm{bip}(G))$. If G is a graph, then we have $G_{[2]} = G$.

For a general hypergraph G we define the *line graph* $\boldsymbol{L(G)}$ as the graph with vertex set $E(G)$ such that two (different) vertices v, w in $L(G)$ (which are hyperedge labels in G) are joined if $\mathcal{E}(G)(v) \cap \mathcal{E}(G)(w) \neq \emptyset$. We have $L(G) = \mathrm{Br}_2(\mathrm{hyp}(G))$ and thus $L(G) = (G^t)_{[2]}$. If G is a graph, then $L(G)$ is the usual line graph of G (with the edges of G as vertices, connected in $L(G)$ if they are adjacent in G).

2.5 Connected hypergraphs

A general hypergraph G is *connected* if from $G_1 = G_1 \cup G_2$ it follows that at least one of G_1 or G_2 is the general null hypergraph. It is G connected if and only if $\mathrm{bip}(G)$ is connected. It follows that G is connected iff G^t is connected.

The empty hypergraph is connected. A general hypergraph G containing an empty hyperedge is connected iff $V(G) = \emptyset$ and $|E(G)| = 1$. If G is a general hypergraph with a singleton edge label $e \in E(G)$, then G is connected iff $G \setminus \{e\}$ is connected.

For every general hypergraph G there is exactly one set S of general non-null hypergraphs such that $G = \bigcup_{G' \in S} G'$, and we define $\mathrm{cc}(G) := S$ as the set of *connected components* of G. Thus G is connected iff $|\mathrm{cc}(G)| \leq 1$.

For vertices or hyperedges $a, b \in V(G) \cup E(G)$ in a general hypergraph G a *walk* from a to b in G is a walk from a to b in $\mathrm{bip}(G)$. Thus we can move from one vertex to another in a general hypergraph iff there is some hyperedge incident with both vertices, and we can move from one hyperedge to another iff both hyperedges have some vertex in common. If G is a graph, then the notion of a walk of G as hypergraph is the "explicit notion" of a walk in a graph, with vertices and edges alternating.

If G has no empty edge, then G is connected iff between any two vertices of G there exists a walk (in other words, iff the 2-section $G_{[2]}$ is connected), and if G has no non-covered vertex, then G is connected iff between any two hyperedges of G there exists a walk (in other words, iff the line graph $L(G)$ is connected).

(For arbitrary general hypergraphs G we have, that if G is connected, then $G_{[2]}$ and $L(G)$ are connected.)

2.6 Separating sets in hypergraphs

A *separating hyperedge set* in a connected general hypergraph G is a set $N \subseteq E(G)$ such that $G \setminus N$ is not connected. A *separating vertex set* in G is a set $M \subseteq V(G)$ such that $G \setminus M$ is not connected. And finally a *separating mixed set* in G is a set $X \subseteq V(G) \cup E(G)$ such that $G \setminus X$ is not connected. N is a separating hyperedge set in G iff N is a separating vertex set in G^t, and M is a separating vertex set in G iff M is a separating hyperedge set in G^t. Finally N is a separating hyperedge set in G iff N is a separating vertex set in $\mathrm{bip}(G)$ with $N \subseteq B(\mathrm{bip}(G))$, M is a separating vertex set in G iff M is a separating vertex set in $\mathrm{bip}(G)$ with $M \subseteq A(\mathrm{bip}(G))$, and X is a separating mixed set in G iff X is a separating vertex set in $\mathrm{bip}(G)$.

A separating vertex set M is called *non-trivial* if not only $G \setminus M$ is disconnected, but also $(G \setminus M)^*$. And a separating hyperedge set in G is called non-trivial if it is a non-trivial separating vertex set in G^t. Thus a non-trivial separating vertex set $M \subseteq V(G)$ for a connected G is characterised by the condition $(G \setminus M)^*$ disconnected, while a non-trivial separating hyperedge set $N \subseteq E(G)$ is characterised by the condition $(G \setminus N)_*$ disconnected.

For every general hypergraph G and for $M \subseteq V(G)$ we have $(G \setminus M)_{[2]} = G_{[2]} - M$. The non-trivial separating vertex sets of connected G are exactly the separating vertex sets of $G_{[2]}$.[2)] Here is the proof for the last fact: M non-trivial separating vertex set in $G \Leftrightarrow (G \setminus M)^*$ disconnected $\Leftrightarrow ((G \setminus M)^*)_{[2]}$ disconnected, where $((G \setminus M)^*)_{[2]} = (G \setminus M)_{[2]} = G_{[2]} - M$.

Every separating hyperedge set of G leads canonically to a separating edge set of $G_{[2]}$, but *not vice versa*: By removing edges from $G_{[2]}$ we can create graphs which cannot be considered as 2-sections of general hypergraphs anymore. Thus there are richer possibilities for separating edge sets in $G_{[2]}$ than for separating hyperedge sets in G.[3)]

For every general hypergraph G and for $N \subseteq E(G)$ we have $L(G \setminus N) = L(G) - N$. The non-trivial separating hyperedge sets of connected G are exactly the separating vertex sets of $L(G)$.[4)] And for every non-trivial separating vertex set in G we have a corresponding separating edge set in $L(G)$ (but not vice versa).[5)]

To summarise: To separate a connected general hypergraph G we either can use separating vertex sets or separating hyperedge sets, and we can also mix both possibilities. These possibilities are exactly reflected by using separating vertex sets in the vertex-hyperedge-graph $\mathrm{bip}(G)$, either only using vertices from the first part, or the second part, or from both parts. Whence the separating vertex sets in G are exactly the separating hyperedge sets in G^t, while the separating hyperedge sets in G are exactly the separating vertex sets in G^t.

Trivial separating vertex sets as well as trivial separating hyperedge sets can be discarded (at least in our context). The non-trivial separating vertex sets in G are exactly the non-trivial separating hyperedge sets in G^t, and vice versa.

Finally, the non-trivial separating vertex sets in G are exactly the separating vertex sets in $G_{[2]}$, and the non-trivial separating hyperedge sets in G are exactly the separating vertex sets in $L(G)$. Considering separating edge sets in $G_{[2]}$ or $L(G)$ is not as useful.

So if (non-trivial) separating vertex sets in G are of interest, then one of

1. (non-trivial) separating vertex sets in G
2. (non-trivial) separating hyperedge sets in G^t
3. (non-trivial) separating vertex sets in the first part of $\mathrm{bip}(G)$
4. separating vertex sets in $G_{[2]}$

[2)] Trivial separating vertex sets can be discarded in all our applications.

[3)] However for all the hypergraphs we consider in our applications, separating edge sets in $G_{[2]}$ which do not correspond to separating hyperedge sets in G have no meaning.

[4)] Trivial separating hyperedge sets can be discarded in all our applications.

[5)] For all the hypergraphs we consider in our applications, separating vertex sets in $L(G)$ which do not correspond to separating hyperedge sets in G have no meaning.

can be considered (equivalently), while if (non-trivial) separating hyperedge sets in G are of interest, then one of

1. (non-trivial) separating hyperedge sets in G
2. (non-trivial) separating vertex sets in G^t
3. (non-trivial) separating vertex sets in the second part of $\mathrm{bip}(G)$
4. separating vertex sets in $L(G)$

can be considered (equivalently). And finally, if (non-trivial) mixed separating vertex sets in G are of interest, then the (equivalent) alternative is to consider (non-trivial) separating vertex sets in $\mathrm{hyp}(G)$.[6)]

We see, that if we start with a (general) hypergraph G, then the splitting possibilities are completely covered by considering separating vertex sets of the three graphs $G_{[2]}$, $L(G)$ and $\mathrm{bip}(G)$, and thus for theoretical convenience we might restrict our attention to separating *vertex* sets in *graphs*.[7)]

3 Applying splittings

We do not study graphs etc. on their own, but for us they always represent some problem, for example a kind of constraint satisfaction problem or a generalised satisfiability problem. So consider such a "problem instance" P, where we have a general notion of "graph representations" $G(P)$ for the whole problem class. The two basic requirements so that $G(P)$ is of any help to us (regarding the general splitting approach) are the following:

(I) To the connected components $G' \in \mathrm{cc}(G(P))$ we can associated "parts" $P(G')$ of G such that these parts together represent a "partitioning" (without overlap) of P, and such that we can solve the problem instances $P(G')$ on their own, and from these combined solutions we can efficiently compute a solution for P.

(II) Given a set $S \subseteq V(G(P))$, we must have the possibility to create "branches" $P_1, \ldots, P_m$, such that from solutions for all the $P_1, \ldots, P_m$ we can efficiently compute a solution for P, and such that $G(P_i)$ is naturally embedded into $G(P) - S$.[8)]

Actually with (I) alone we can already make good use of $G(P)$ by just passively reacting to observed splittings in some procedure solving P, however on this approach there is not much to say from the general point of view, and (II) is needed so that some machinery from graph theory can be applied.

[6)] A mixed set X is separating in G if $(G \setminus X)^*_*$ is disconnected. A general hypergraph with $G^*_* = G$ (i.e., a general hypergraph without empty hyperedges or non-covered vertices) is also called a "Berge-Hypergraph".

[7)] And if we start with a graph G, then we either consider the separating vertex sets in G (obviously), or we can consider the separating vertex sets in $L(G)$ (capturing the non-trivial separating edge sets), or we can consider the separating vertex sets in $\mathrm{bip}(G)$ (capturing the non-trivial separating mixed sets); we have used the implicit promotion of G to a hypergraph for the last two possibilities.

[8)] That is, there is a natural injection $j : V(G(P_i)) \to V(G(P) - S)$ such that j maps edges of $G(P_i)$ to edges of $G(P) - S$ (and thus $G(P_i)$ can naturally be associated with a subgraph of $G(P) - S$).

A general approach, based on splitting graph representations, yielding a recursive procedure $\mathcal{GS}(P)$ (like "graph splitting") solving P, can be outlined as follows (using several graph representation $G_1, \ldots, G_p$; although we can combine vertex and edge splitting into one graph, in general there doesn't seem to be a good possibility to combine really different graph representations):

1. If we find one $G_i(P)$ with at most one vertex (so that there are no separating vertex sets), then we should have an efficient algorithm for solving P.
2. If we find one disconnected $G_i(P)$, then we can compute $\mathcal{GS}(P(G'))$ for all $G' \in \mathrm{cc}(G(P))$, and combining all these solutions into a solution for P.
3. Choose $i \in \{1, \ldots, p\}$ and a separating vertex set S in $G_i(P)$.
4. Choose branches $P_1, \ldots, P_m$ belonging to S.
5. Compute $\mathcal{GS}(P_i)$ for $i \in \{1, \ldots, m\}$, and combine these solutions into a solution for P.

Remarks regarding the expected time complexity of this approach:

1. If we can make sure that at least two of the connected components of $G(P)$ are of reasonable size, then the recursive calls of $\mathcal{GS}$ in Step 2 are harmless.
2. The recursive calls needed in Step 5 are the problem of this approach.
3. The graph representations $G_i(P)$ needs to be computed from scratch only for the input, while for all other graph representations the natural embeddings can be used.

General strategies for finding a good separating vertex set in Step 3, which yield some global performance guarantee:

1. If $G(P)$ is planar, then the planar separator theorem ([28]) can be applied as in [29]: This theorem guarantees the existence (and easy computation) of a separating vertex set C for every planar graph G with $|C| \leq 2\sqrt{2}\sqrt{|V(G)|}$ splitting G into two pieces each having at most $\frac{2}{3}|V(G)|$ many vertices.
2. The theory of tree decompositions yields methods for finding global splitting strategies; see Section 5.

We do not see how with this (general) approach FPT can be achieved; it seems that for FPT some "intelligent control" is needed, basically avoiding the recursive calls in Step 5 by strengthening the recursive calls in Step 2.

Finally, the field of hypergraph partitioning algorithms provides many optimisation algorithms for (heuristically) finding good separating hyperedge sets, where typically the size of the set can be optimised, while via constraints the size of the resulting components can be restricted ([34] seems to be a good starting point).

4 Heuristical evaluation

The fundamental question with the generic scheme $\mathcal{GS}$ is how to find "good" separating vertex sets S in Step 3 ? In this section we outline a generic algorithm $\mathcal{HS}$ (for "heuristical splitting") refining $\mathcal{GS}$, where we make the simplifying assumption that we consider only one graph representation G (integrating several graph representations poses the problem of how to compare measurements taken from different representations, and at this time we do not have a reasonable understanding of that.)

We take the point of view (following [22, 26, 27]) that we have given a set of possibilities, and we have to choose one (the "best candidate" according to our evaluation criterion). Since the branches $P_1, \ldots, P_m$ calculated in Step 4 are determinative, exactly they need to be considered in the evaluation, and so we consider a set $\mathfrak{S}$ of "splits", which are tuples $\mathcal{S} = (P_1, \ldots, P_m)$ where $P_1, \ldots, P_m$ are the branching instances from Step 4 (note that m depends on $\mathcal{S}$).

First there is the possibility that when computing $\mathfrak{S}$ the problem could already been solved, or a reduction was found (simplifying the problem); we assume in the sequel that these cases have been checked and do not apply. For $\mathcal{S} \in \mathfrak{S}$ with $\mathcal{S} = (P_1, \ldots, P_m)$ let $B(\mathcal{S})$ be the set of all $P(G')$ for $G' \in \mathrm{cc}(G(P_i))$, $i \in \{1, \ldots, m\}$, with $|V(G')| \geq 2$. The problems in $B(\mathcal{S})$ have to be all solved, and out of these "partial solutions" a solution for P is computed. So the general evaluation approach is to use some estimation of run-time needed to process the elements of $B(\mathcal{S})$, summing it up, and obtaining a positive real number estimating the total run-time (which is to be minimised).[9)]

In our problem domain the basic approach for estimating run-times is to use some base $b \in \mathbb{R}_{>1}$, to fix a complexity measurement $\mu(P) \in \mathbb{R}_{\geq 0}$ for problems P, and estimate the run-time as $b^{\mu(P)}$. So the evaluation of a split $\mathcal{S} \in \mathfrak{S}$ is

$$\sum_{P' \in B(\mathcal{S})} b^{\mu(P')}$$

and to choose a split $\mathcal{S} \in \mathfrak{S}$ where this evaluation is minimised.

This leaves the problems of choosing μ and b. The simplest (natural) choice for $\mu(P)$ here is $|V(G(P))|$ (the number of vertices in the graph representation). For the choice of b we propose the following dynamic scheme:

We start with some initial value $b := b_0$ (for example $b_0 = 2$). And after a split $\mathcal{S}$ has been selected, we compute the branching tuple t (a vector of positive real numbers) of width $|B(\mathcal{S})|$ (note that due to our assumption, that no reduction is applicable, we have $|B(\mathcal{S})| \geq 2$), where the component value corresponding to branch $P' \in B(\mathcal{S})$ is $b^{\mu(P)} - b^{\mu(P')}$. We must have here that $\mu(P') < \mu(P)$ (recall that in this context P is the current input of the recursive algorithm $\mathcal{GS}$), which is guaranteed for example with the choice of μ as the number of vertices in the graph representation. Now the new base (used in the recursive calls of $\mathcal{GS}$) is

$$b := \tau(t),$$

using the τ-function as explored in [22, 26, 27], which yields a canonical method for estimating size of (branching) trees.[10)]

[9)] Since $B(\mathcal{S})$ is a set we are able to detect repetition of branches; if however this is too costly then $B(\mathcal{S})$ should be implemented as a list.

[10)] For $t = (t_1, \ldots, t_s)$ it is $\tau(t) > 1$ the unique solution of $\sum_{i=1}^{s} x^{-t_i} = 1$. We can compute $\tau(t)$ efficiently using Newton's method.

5 Forest and tree decompositions of graphs

The main purpose of this section is to introduce the parts of the theory of tree decompositions of graphs needed in our context (see Sections 10.4 and 10.5 in [19] for a readable introduction). We will introduce the formal extension of the notion of tree decomposition to a *forest decomposition*, which allows for a more elegant representation. In Subsection 5.2 we will exploit tree decomposition "as is" (without using further problem structure), obtaining general schemes from them for splitting a graph into small components (this approach, using tree decompositions "top down", not as usual "bottom up", was motivated by a talk with Stephan Kreutzer on the Logic Colloquium 2005 in Athens). Finally, in Subsection 5.3 we apply these splitting trees to the generic algorithm $\mathcal{GS}$, obtaining algorithm $\mathcal{TS}$ which runs in polynomial time in the treewidth (under certain quite general conditions).

5.1 Forest decompositions

A **forest decomposition** of a hypergraph G is a pair $(\mathcal{F}, \chi)$, where $\mathcal{F}$ is a forest and χ is a set-valued map with $V(\mathcal{F}) \subseteq \mathrm{dom}(\chi)$ such that

(i) for every edge $e \in E(G)$ there exists a vertex $t \in V(\mathcal{F})$ with $e \subseteq \chi(t)$;

(ii) for every vertex $v \in V(G)$ the induced subgraph $\mathcal{F}_v$ of $\mathcal{F}$ given by all nodes $t \in V(\mathcal{F})$ with $v \in \chi(t)$ is connected.

If we have a family $((\mathcal{F}_i, \chi_i))_{i \in I}$ of forest decompositions for a hypergraph family $(G_i)_{i \in I}$, then we say that $((\mathcal{F}_i, \chi_i))_{i \in I}$ is *disjoint* if the set family $(V(\mathcal{F}_i))_{i \in I}$ is disjoint. If $(\mathcal{F}, \chi)$ is a forest decomposition of (some) G and $(\mathcal{F}', \chi')$ is a forest decomposition of (some) G', then we write $(\mathcal{F}, \chi) \subseteq (\mathcal{F}', \chi')$ (and say that $(\mathcal{F}, \chi)$ is contained in $(\mathcal{F}', \chi')$) if $\mathcal{F} \subseteq \mathcal{F}'$, $\mathrm{dom}(\chi) \subseteq \mathrm{dom}(\chi')$ and $\forall\, v \in \mathrm{dom}(\chi) : \chi(v) \subseteq \chi'(v)$.

Compared with the usual notion of "tree decomposition", we allow for the following (convenient, but finally inessential) extensions:

1. $\mathcal{F}$ does not need to be connected (and may be empty).
2. The domain of χ may contain also other elements than those in $V(\mathcal{F})$.
3. For $v \in V(G)$ the set $\chi(v)$ may contain elements not in $V(G)$.
4. Isolated vertices of G may not be covered by $(\mathcal{F}, \chi)$.

A *tree decomposition* is a forest decomposition $(\mathcal{F}, \chi)$ where $\mathcal{F}$ actually is a tree. The *width* of a tree decomposition (T, χ) is $\min_{v \in V(T)} |\chi(t)| - 1$, while the *tree width* $\mathrm{tw}(G)$ of a (connected) G is the minimal width over all tree decompositions of G (note that every hypergraph G has a trivial tree decomposition, where the tree just contains one node, and χ assigns to this node the set $V(G)$; thus $\mathrm{tw}(G) \leq |V(G)|$ - 1).[11)]

If $(\mathcal{F}, \chi)$ is a forest decomposition of G, then by the following operations we obtain simpler forest decompositions $(\mathcal{F}', \chi')$ of G:

S1 In case of $V(\mathcal{F}) \subset \mathrm{dom}(\chi)$ or there is $v \in V(\mathcal{F})$ with $\chi(v) \not\subseteq V(G)$ set $\mathcal{F}' := \mathcal{F}$ and let χ' have domain $V(\mathcal{F})$ with $\chi'(t) := \chi(t) \cap V(G)$ for $t \in V(\mathcal{F})$.

[11)] We have $\mathrm{tw}(G) = -1$ iff G contains no non-empty hyperedges, while otherwise we have $\mathrm{tw}(G) \geq 0$. The notion of tree width is only sensible for connected G (see Corollary 5.4).

S2 For $t \in V(\mathcal{F})$ with $\chi(t) = \emptyset$ set $\mathcal{F}' := \mathcal{F} - \{t\}$ and $\chi' := \chi$.

S3 For $\{t, t'\} \in E(\mathcal{F})$ with $\chi(t) \subseteq \chi(t')$ let $\chi' := \chi$ and obtain $\mathcal{F}'$ from $\mathcal{F}$ by removing vertex t and reconnecting all neighbours x of t different from t' to t' (formally $\mathcal{F}'$ has vertex set $V(\mathcal{F}') := V(\mathcal{F} - \{t\})$ and edge set $E(\mathcal{F}') := E(\mathcal{F} - \{t\}) \cup \{\{x, t'\} : \{x, t\} \in E(G) \wedge x \neq t'\}$).

A forest resp. tree decomposition of G such that none of these simplifications can be applied anymore is called *irredundant*.

Lemma 5.1 *Consider a hypergraph G.*

1. *From a forest decomposition $(\mathcal{F}, \chi)$ of G in polynomial time an irredundant forest decomposition $(\mathcal{F}', \chi')$ of G with $V(\mathcal{F}') \subseteq V(\mathcal{F})$ and $\forall\, v \in V(\mathcal{F}') : \chi'(v) \subseteq \chi(v)$ can be obtained.*
2. *If $(\mathcal{F}, \chi)$ is an irredundant forest decomposition of G, then $|V(\mathcal{F})| \leq |V(G)|$ holds.*

Proof: Part 1 is achieved by simply applying the simplification rules as long as possible. Part 2 is proven by induction on $|V(G)|$:

If $V(G) = \emptyset$, then the only irredundant forest decomposition is $((\emptyset, \emptyset), \emptyset)$. So assume $v \in V(G)$. Compare (10.16) in [19]. ∎

The following basic properties of forest decompositions $(\mathcal{F}, \chi)$ of graphs G regarding subgraph formation and graph union follow directly from the definitions (these properties prompted us to introduce the notion of a *forest* decomposition):

I If G' is a subgraph of G, then $(\mathcal{F}, \chi)$ is a forest decomposition of G'.

II If $\mathcal{F}'$ is an induced subgraph of $\mathcal{F}$, then $(\mathcal{F}', \chi)$ is a forest decomposition of the induced subgraph of G given as $G - \bigcup_{t \in V(\mathcal{F}) \setminus V(\mathcal{F}')} \chi(t)$.

III If $\mathcal{F}'$ is a partial graph of $\mathcal{F}$, then $(\mathcal{F}', \chi)$ is a forest decomposition of the induced subgraph of G given as $G - \bigcup_{\{u,v\} \in E(\mathcal{F}) \setminus E(\mathcal{F}')} (\chi(u) \cap \chi(v))$.

IV If (G_i) is a family of vertex-disjoint graphs, and for each $i \in I$ we have an irredundant forest decomposition $(\mathcal{F}_i, \chi_i)$ of G_i, then $(\bigcup_{i \in I} \mathcal{F}_i, \bigcup_{i \in I} \chi_i)$ is an irredundant forest decomposition of $\bigcup_{i \in I} G_i$.

Now the central lemma:

Lemma 5.2 *Consider a forest decomposition $(\mathcal{F}, \chi)$ of a graph G and vertices $t, t' \in V(\mathcal{F})$. If t, t' are not connected in $\mathcal{F}$, then no vertices $v, v' \in V(G)$ with $v \in \chi(t)$ and $v' \in \chi(t')$ are connected in G neither.*

Proof: We prove by induction that for all disconnected nodes t, t' of $\mathcal{F}$ there is no path of length l from v to v' in G for any $v \in \chi(t)$, $v' \in \chi(t')$. If $l = 0$ then $v = v'$, and thus $t, t' \in V(\mathcal{F}_v)$ where $\mathcal{F}_v$ by (ii) is connected contradicting the assumptions that t, t' are in different connected components of $\mathcal{F}$. So assume $l > 0$. There exists now a neighbour v'' of v in G such that a path of length $l - 1$ from v'' to v' exists. By (i) there is $t'' \in V(\mathcal{F})$ with $v, v'' \in \chi(t'')$. Thus $t'' \in V(\mathcal{F}_v)$, whence t'', t' are disconnected in $\mathcal{F}$ contradicting the induction assumption. ∎

Corollary 5.3 *If $(\mathcal{F}, \chi)$ is an irredundant forest decomposition of a connected graph G, and $V(\mathcal{F}) \neq \emptyset$, then $(\mathcal{F}, \chi)$ is an irredundant tree decomposition of G.*

Corollary 5.4 *If $(\mathcal{F}, \chi)$ is a forest decomposition of G, and G' is a connected component of G with at least two vertices, then there is exactly one connected component $\mathcal{F}'$ of $\mathcal{F}$ with $V(\mathcal{F}') \cap V(G') \neq \emptyset$, and $(\mathcal{F}', \chi)$ is a tree decomposition of G'.*

That we have suspended considering forest decompositions of *hyper*-graphs half way through (in order to avoid having to define sub-hypergraphs etc.) can be justified as follows:

Lemma 5.5 *Consider a forest decomposition $(\mathcal{F}, \chi)$ of a graph G. If $C \subseteq V(G)$ is a clique in G (all different vertices in C are adjacent) with $|C| \geq 2$, then there must be $v \in V(\mathcal{F})$ with $C \subseteq \chi(v)$.*

Proof: XXX

Corollary 5.6 *Consider a hypergraph G. The forest decompositions of G are exactly the forest decompositions $(\mathcal{F}, \chi)$ of $G_{[2]}$ which fulfil the following additional (trivial) requirements (these are empty if G does not have hyperedges of size smaller than two):*

1. *If G contains the empty hyperedge then $\mathcal{F}$ shall have vertices.*
2. *If G contains a singleton hyperedge $\{v\}$, then there must be a vertex $t \in V(\mathcal{F})$ with $v \in \chi(t)$.*

Corollary 5.7 *Consider a connected hypergraph G.*

1. *If $E(G) \neq \emptyset$, then the treewidth of G (which except of trivial cases equals the treewidth of $G_{[2]}$) is at least as big as the maximal hyperedge size in G.*
2. *If $V(G) \neq \emptyset$, then the treewidth of G^t (which except of trivial cases equals the treewidth of $L(G)$) is at least as big as the maximal vertex degree in G.*

5.2 Splitting trees

Consider a connected graph G and an irredundant tree decomposition (T, χ) of G. We want to construct a rooted tree $\mathcal{T}(G, (T, \chi))$ describing the process of breaking G into smaller and smaller pieces by recursive splits of T. The splitting tree $\mathcal{T}(G, (T, \chi))$ shall fulfil the following requirements:

(i) Every node is labelled by a triple $(G', (T', \chi'), r')$, where G' is a connected subgraph of G, (T', χ') is an irredundant tree decomposition of G', contained in (T, χ), and $r' \in V(T')$.

(ii) The subgraphs labelling the children of an inner node labelled with the triple $(G', (T', \chi'), r')$ are exactly the connected components of $G' - \chi(r')$, while the subgraph labelling the root is just G.

(iii) The tree decompositions labelling the children of an inner node labelled with the triple $(G', (T', \chi'), r')$ are contained in (T', χ') and are pairwise disjoint. The tree decomposition labelling the root is just (T, χ).

(iv) Exactly the tree decompositions labelling the leaves are trivial (have a tree with only one node).

(v) The height of $\mathcal{T}(G,(T,\chi))$ is at most $\lceil \log_2(|V(T)|)\rceil$.

In order to fulfil (v), we need the following simple observation.

Recall that a *rooted tree* is a pair (T,r) where T is a tree and $r \in V(T)$. Given a rooted tree (T,r) and a vertex $r' \in V(T)$, we denote by $T_{r'}^r$ the induced subgraph of T given by all vertices $v \in V(T)$ such that r' is part of the (unique) path from r to v in T. The *children* of the root r in (T,r) are all vertices adjacent to r in T.

Lemma 5.8 *Consider a tree T. There exists a vertex $r \in V(T)$ such that for the rooted tree (T,r) and every child r' of r in (T,r) we have $|V(T_{r'}^r)| \leq \frac{1}{2}|V(T)|$.*

Proof: If $|V(T)| = 1$, then the assertion is trivial, and so assume $|V(T)| \geq 2$. For a vertex $v \in V(T)$ let

$$m(v) := \max_{v' \in V(T)\setminus\{v\}} |V(T_{v'}^v)|.$$

Choose some $r_0 \in V(T)$ and consider (T,r_0). If $m(r_0) \leq \frac{1}{2}|V(T)|$, then choose $r := r_0$. Otherwise there exists a vertex r_1 adjacent to r in T, such that we have $|E(T_{r_1}^{r_0})| > \frac{1}{2}|E(T)|$. So consider (T,r_1). We have $m(r_1) \leq m(r_0) - 1$. Iterating the process we must finally obtain a suitable root. ∎

Now the (straight-forward) construction of $\mathcal{T}(G,(T,\chi))$ is as follows:

1. If $V(T) = \{r\}$, then $\mathcal{T}(G,(T,\chi))$ is just one node labelled with $(G,(T,\chi),r)$. So assume $|V(T)| \geq 2$.
2. Choose a "root vertex " $r \in V(T)$ fulfilling the criterion of Lemma 5.8. Now the root of $\mathcal{T}(G,(T,\chi))$ is labelled with $(G,(T,\chi),r)$. Let $k \geq 1$ be the degree of r in T.
3. Consider the forest $\mathcal{F} := T - \{r\}$, which has (exactly) k connected components T_i for $i \in \{1,\ldots,k\}$. According to II it is $(\mathcal{F},\chi)$ a forest decomposition of $G' := G - \chi(r)$.
4. Let $m := |\mathrm{cc}(G')| \geq k$. It is $(\mathcal{F},\chi)$ a forest decomposition of G'' for all $G'' \in \mathrm{cc}(G')$ by I. There will be exactly m subtrees hanging at the root of $\mathcal{T}(G,(T,\chi))$.
5. For every $G'' \in \mathrm{cc}(G')$ obtain an irredundant tree decomposition (T'',χ'') from $(\mathcal{F},\chi)$ (applying Corollary 5.3), and add a subtree $\mathcal{T}(G'',(T'',\chi''))$ to the root of $\mathcal{T}(G,(T,\chi))$.

Remarks:

1. The construction of $\mathcal{T}(G,(T,\chi))$ is based on "splitting on nodes of the tree decomposition", using property II, and exploiting Lemma 5.8. If we want to use "splitting on edges of the tree decomposition" (using property III), then we need to restrict the degree of nodes in T.
2. The construction is based on just one tree decomposition (computed for the input), which can be motivated by the fact that for every given (fixed) $k \in \mathbb{N}_0$, if we know that $\mathrm{tw}(G) \leq k$ holds, then we can compute a tree decomposition of width at most k in linear time; however, the constants for "linear time" here are very large (and depend on k). Deciding whether G has a tree decomposition of width at most k, where k is part of the input to the decision

problem, in NP-complete (see [5] for these complexity results). So in practice approximation algorithms are used, and then it might we worth recomputing the tree decompositions (hopefully obtained better decompositions) for the graphs G'' in Step 5.

5.3 Applying tree decompositions

The basic approach is to specialise the general approach $\mathcal{GS}$ from Section 3 (using now only one graph representation $G_1 = G$). Consider an irredundant tree decomposition (T, χ) of $G(P)$ together with a splitting tree $\mathcal{T}(G(P), (T, \chi))$. We simply synchronise the branching tree built up by algorithm $\mathcal{GS}$ with the splitting tree $\mathcal{T}(G(P), (T, \chi))$, and choose in Step 3 of $\mathcal{GS}$ always $S = \chi'(r')$ for the currently considered tree decomposition (T', χ'). Since for the construction of splitting trees we assume the graph to be connected, if $G(P)$ for the input is disconnected then first a branching according to Step 2 of $\mathcal{GS}$ is to be performed.

Let us call this specialised (generic) algorithm $\mathcal{TS}(P)$ (like "tree splitting"). The time complexity of $\mathcal{TS}(P)$ in terms of nodes in the branching tree can be estimated as follows:

Let k be the width of (T, χ), and assume that in Step 4 of $\mathcal{GS}$ we always have $m \leq f(k, P)$. Then the number of nodes in the branching tree belonging to $\mathcal{GS}$ can be estimated as follows (using Lemma 5.1, Part 2, and the estimation, that for branching in Step 2 of $\mathcal{GS}$ the case of just two connected components is the worst case):

$$\begin{aligned}&\#\text{nds} \leq (2 \cdot f(k,P))^{\lceil \log_2(|V(T)|)\rceil+1} - 1 \leq (2 \cdot f(k,P))^{\log_2(|V(G)|)} = \\ &|V(G)| \cdot f(k,P)^{\log_2(|V(G)|)} = |V(G)| \cdot |V(G)|^{\log_2(f(k,P))} = |V(G)|^{\log_2(f(k,P))+1}\end{aligned}$$

$\mathcal{TS}(P)$ provides a simple, but general exploit of tree decompositions by recursively branching, yielding solution of P in polynomial time in the treewidth k if the branching bound $f(k, P)$ is only exponential in k. This is weaker than the "dynamic programming" refinement discussed below, but can be applied in general, and also allows easy integration with special reasoning for the concrete problem structure (since the P_i in Step 4 of $\mathcal{GS}$ can depend on the current problem instance itself).

In order to obtain FPT we cannot afford branchings depending on k or P (as above); the best (affordable) techniques are based on "dynamic programming" approaches, which explore special properties of the graph representation, and are not discussed further in this report (however it is a very interesting and important question to find out more abstract foundations of these (feasible) approaches).

6 Special graph representations of (labelled) clause-sets

We consider specific graph and hypergraph representations $G(F)$ of labelled clause-sets F here, where by using *labelled* clause-sets we avoid the problems associated with contraction of clauses in clause-*sets* when applying partial assignments.

6.1 The variable hypergraph, and derived graphs

Let $\mathbf{vhg(}F\mathbf{)}$ be the *variable hypergraph* of F, which is a general hypergraph with vertex set $\mathrm{var}(F)$, hyperedge label set $L(F)$ and hyperedge function $l \mapsto \mathrm{var}(F(l))$ (in other words, the variable hypergraph results from F by just dropping the polarities from the literals). Thus we have $|V(\mathrm{vhg}(F))| = n(F)$ and $|E(\mathrm{vhg}(F))| = c(F)$.

To a connected component $G' \in \mathrm{cc}(\mathrm{vhg}(F))$ we associated the sub-labelled-clause-set $F(G') \subseteq F$ specified by the condition $\mathrm{vhg}(F(G')) = G'$. We then have

$$\begin{aligned} F &= \dot{\bigcup}_{G' \in \mathrm{cc}(\mathrm{vhg}(F))} F(G') \\ \mathrm{mod_t}(F) &= \circ_{G' \in \mathrm{cc}(\mathrm{vhg}(F))} \mathrm{mod_t}(F(G')) \\ |\mathrm{mod_t}(F)| &= \prod_{G' \in \mathrm{cc}(\mathrm{vhg}(F))} |\mathrm{mod_t}(F(G'))|. \end{aligned}$$

The representation of F as a graph with bipartition (or an incidence structure) is given by the *variable-clause graph* $\mathbf{vcg(}F\mathbf{)} := \mathrm{bip}(\mathrm{vhg}(F))$ (also "incidence graph" ([35])).

For a partial assignment φ we have a natural embedding of $\mathrm{vcg}(\varphi * F)$ into $\mathrm{vcg}(F)$, since by definition we actually have $\mathrm{vcg}(\varphi * F) \subseteq \mathrm{vcg}(F)$.[12)]

Via the standard constructions of 2-sections and line graphs we obtain:

1. The *variable-interaction graph* $\mathbf{vig(}F\mathbf{)} := \mathrm{vhg}(F)_{[2]}$ (also called "primal graph" ([15, 35]) or "interaction graph" ([31]), or "Gaifman graph" ([8])) has the variables of F as vertices, which are joined by an edge if there exists a clause containing both variables. It is $\mathrm{vig}(\varphi * F)$ a subgraph of $\mathrm{vig}(F)$ (in general not induced).
2. The *common-variable graph* $\mathbf{cvg(}F\mathbf{)} := L(\mathrm{vhg}(F))$ has the clause-labels as vertices, where two (different) clauses C, D are joined by an edge if $\mathrm{var}(C) \cap \mathrm{var}(D) \neq \emptyset$. It is $\mathrm{cvg}(\varphi * F)$ a subgraph of $\mathrm{cvg}(F)$ (in general not induced).

Regarding splitting, we have the standard three choices:

1. Splitting on variables, i.e., considering separating vertex sets in $\mathrm{vhg}(F)$ or $\mathrm{vig}(F)$, or considering separating hyperedge sets in $\mathrm{vhg}(F)^t$.
2. Splitting on clauses, i.e., considering separating vertex sets in $\mathrm{cvg}(F)$ or $\mathrm{vhg}(F)^t$, or considering separating hyperedge sets in $\mathrm{vhg}(F)$.
3. Splitting on mixed sets of variables and clauses, i.e., considering separating vertex sets in $\mathrm{vcg}(F)$.

The hypergraph choices are superior regarding efficiency, but for the sake of (theoretical) convenience we consider the graphs here (w.l.o.g.), that is, we consider separating vertex sets in $\mathrm{vig}(F)$, $\mathrm{cvg}(F)$ and $\mathrm{vcg}(F)$. The most common choice are separating vertex sets in $\mathrm{vig}(F)$, while separating vertex sets in $\mathrm{cvg}(F)$ are inferior to separating vertex sets in the conflict graph and refinements as studied in Subsection 6.2; finally separating vertex sets in $\mathrm{vcg}(F)$ seem to offer interesting possibilities ([35, 32]; see the remark below) but yet a better understanding is needed.

[12)] This motivates the use of *labelled* clause-sets.

Regarding separating vertex sets in $\mathrm{vig}(F)$, we can identify the following directions of research:

- A direct application of the planar separator theorem ([28]) as in [29] (Section 3 "Nonserial dynamic programming") yields that in case of planar $\mathrm{vig}(F)$ we have SAT decision (and computation of MAXSAT) in time $2^{O(\sqrt{n(F)})}$. This result has been generalised in [18], using the "crossing number" of a graph to measure the degree of planarity.
- An overview of various general decomposition techniques is given in [14] (with a focus on providing polynomial-time guarantees).

Regarding tree decomposition, it is well known that SAT decision for F is FPT in the treewidth of $\mathrm{vig}(F)$. In [35] it was shown that also SAT decision for F is FPT in the treewidth of $\mathrm{vcg}(F)$, but relying on practically infeasible methods. Recently in [32] a direct and more efficient algorithm has been given.

6.2 The conflict graph and refinements

The remaining graph representations $G(F)$ of clause-sets are all partial graphs of $\mathrm{cvg}(F)$ (so they are "clause-based", that is, there vertex set is the set of clause-labels of F). To a connected component G' of $G(F)$ we will associate $F(G') \subseteq F$ with $L(F(G')) = V(G')$, while $\mathrm{var}(F(G'))$ is the set of all variables occurring in some clause of G'. If F does not contain uncovered variables (variables which do not occur in any clause of F), then we have

$$F = \bigcup\!\!\!\!\cdot_{G' \in \mathrm{cc}(G(F))} F(G').$$

Now the three specialisations are as follows:

- The **conflict graph $\mathrm{cg}(F)$** (sometimes also called "resolution graph", but we reserve this notion for the proper resolution graph below) has an edge joining two clauses C, D if they clash in at least one literal. It is $\mathrm{cg}(\varphi * F)$ an *induced* subgraph of $\mathrm{cg}(F)$.
- The **resolution graph $\mathrm{rg}(F)$** has an edge joining two clauses C, D if they clash in *exactly* one literal. It is $\mathrm{rg}(\varphi * F)$ an *induced* subgraph of $\mathrm{rg}(F)$.
- The **subsumption-resolution graph $\mathrm{srg}(F)$** has an edge joining two clauses C, D if they clash in exactly one literal and if the resolvent is not subsumed by a clause in F. It is $\mathrm{srg}(\varphi * F)$ a subgraph of $\mathrm{srg}(F)$ (in general not induced).

By definition we have $E(\mathrm{cvg}(F)) \supseteq E(\mathrm{cg}(F)) \supseteq E(\mathrm{rg}(F)) \supseteq E(\mathrm{srg}(F))$. Figure 1 illustrates these representations.

In [20] (Lemma 15) is has been shown that if the subsumption-resolution graph splits into components then for SAT decision we can consider the components on their own; we present the proof here since this results seems not to be known. Call a (finite) family $(F_i)_{i \in I}$ of labelled clause-sets with disjoint clause-label-sets (while variables may be shared) *sr-disconnected* if $\mathrm{srg}(\bigcup_{i \in I} F_i) = \bigcup_{i \in I} \mathrm{srg}(F_i)$ holds (that is, if for all $i, j \in I$, $i \neq j$, and for all resolvable clauses $C \in F_i$, $D \in F_j$ there exists $k \in I$ and $R \in F_k$ with $R \subseteq C \diamond D$).

$C_1 := x_1 \vee x_2 \vee x_3$ $C_2 := \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$ $C_3 := x_1 \vee x_2 \vee x_4$

$C_4 := x_1 \vee x_2 \vee \overline{x_4}$ $C_5 := x_1 \vee \overline{x_3} \vee x_4$ $C_6 := x_2 \vee \overline{x_3} \vee x_4$

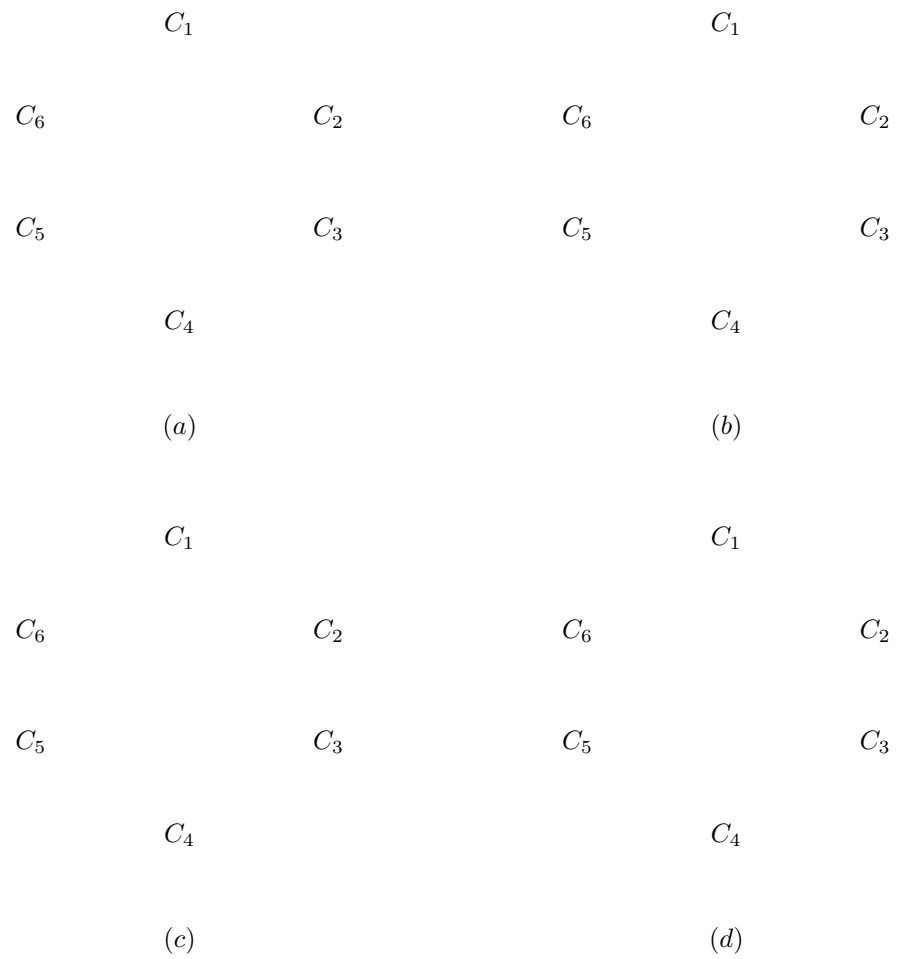

Figure 1: Four clause-based representations of clause-set $\{C_1, \ldots, C_6\}$: (a) common-variable graph; (b) conflict graph; (c) resolution graph; and (d) subsumption-resolution graph.

Recall the definition of the DP-operator $\mathrm{DP}_v(F)$ for a clause-set F and a variable F, which replaces all clauses containing variable v by their (non-tautological) resolvents on v (i.e., $\mathrm{DP}_v(F) = \{C \diamond D : C, D \in F \wedge C \cap \overline{D} = \{v\}\} \cup \{C \in F : v \notin \mathrm{var}(C)\}$). This definition is easily transferred to labelled clause-sets, where as the label of the resolvent of parent clauses C, D with labels c, d we use $\{c, d\}$, and where we remove variable v from the set of variables.

Lemma 6.1 *If the family $(F_i)_{i\in I}$ of labelled clause-sets is sr-disconnected, then also for any variable the family $(\mathrm{DP}_v(F_i))_{i\in I}$ is sr-disconnected.*

Proof: For the proof we apply the well-known "distributivity property" of resolution (see Lemma 7.2 in [27]) in order to commute the resolution steps. ∎

Since for every (labelled) clause-set F we have that F is unsatisfiable iff $\mathrm{DP}_v(F)$ is unsatisfiable, where $\mathrm{DP}_v(F)$ contains one variable less, by induction on $n(\bigcup_{i\in I} F_i)$ we immediately obtain:

Corollary 6.2 *Consider an sr-disconnected family $(F_i)_{i\in I}$ of labelled clause-sets. Then $\bigcup_{i\in I} F_i$ is unsatisfiable if and only if there exists some $i \in I$ such that F_i is unsatisfiable.*

By definition for every labelled clause-set F the family $(F(G'))_{G'\in \mathrm{cc}(\mathrm{srg}(F))}$ is sr-disconnected, whence

Corollary 6.3 *A labelled clause-set F is unsatisfiable if and only if there exists $G' \in \mathrm{cc}(\mathrm{srg}(F))$ such that $F(G')$ is unsatisfiable.*

Open problems:

1. It seems to us that the following holds: "If T a regular resolution refutation of F (see [25]) then the axioms of T all must come from one connected component of $\mathrm{rg}(F)$." The proof should work by induction on T. This would somehow confirm that SAT algorithms (especially with some form of "intelligent backtracking" and focus on recently learned clauses) will tend to work in one component. And it could show a way to attack FPT in the treewidth of the resolution graph (in the spirit of [1]).
2. Given satisfying assignments for all the $F(G')$ in Corollary 6.3, how can we construct a satisfying assignment for F ?!

6.3 Splitting conflict and resolution graphs

The obvious choice for splitting $\mathrm{cg}(F)$ and its refinements $\mathrm{rg}(F)$, $\mathrm{srg}(F)$ is splitting on clauses, that is considering separating vertex sets; these splitting possibilities are definitely better than those in $\mathrm{cvg}(F)$, and their potential waits for exploration (we will go into more algorithmic details in Subsection 7.2).

The second interesting possibility is to consider separating edge sets in $\mathrm{cg}(F)$, $\mathrm{rg}(F)$ and $\mathrm{srg}(F)$. As it was explained in general in Section 2.6, separating edge sets in $\mathrm{cvg}(F)$ (recall that $\mathrm{cvg}(F)$ is the line graph of the variable hypergraph) yield in principal more splitting possibilities than separating vertex sets in $\mathrm{vig}(G)$, however these possibilities seem artificial and vain. Fortunately with the advent of $\mathrm{cg}(F)$, $\mathrm{rg}(F)$ and $\mathrm{srg}(F)$, which remove edges from $\mathrm{cvg}(F)$, separating edge sets become potentially much more interesting; they cover all splitting possibilities given by separating vertex sets in $\mathrm{vig}(F)$, and they embody more structural information about F.[13)] Since the edges of $\mathrm{cg}(F)$ and refinements are naturally labelled by variables (the conflict variables or the resolution variables), splitting on edges in $\mathrm{cg}(F)$ and refinements can happen by splitting on variables; now when splitting on variables then in general actually not one edge but many edges are removed,

[13)] Can one draw the conclusion here, that the treewidth of the line graph of $\mathrm{cg}(F)$ (as well as $\mathrm{rg}(F)$ and $\mathrm{srg}(F)$) is at most as large as the treewidth of $\mathrm{vig}(F)$? Seems too optimistic, but there should be something in this direction.

and so it seems crucial for an efficient use of $\mathrm{cg}(F)$ and refinements to exploit this "superstructure".

The edge set of $\mathrm{cvg}(F)$ has a (build-in) superstructure in the form of a special partitioning, namely we have $\mathrm{cvg}(F) = \bigcup_{v \in \mathrm{var}(F)} K_v$ for edge-disjoint complete graphs K_v, given as the induced subgraphs of $\mathrm{cvg}(F)$ given by all clauses $C \in F$ with $v \in \mathrm{var}(C)$. Now $\mathrm{cg}(F)$ has a more interesting superstructure, again as a partitioning of the edge set, namely we have $\mathrm{cg}(F) = \bigcup_{v \in \mathrm{var}(F)} B_v$ for edge-disjoint complete bipartite graphs B_v, given as the induced subgraphs of $\mathrm{cg}(F)$ given (again) by all clauses $C \in F$ with $v \in \mathrm{var}(C)$. Such biclique partitionings have been exploited for the satisfiability domain in [24, 10], and we believe that we have here a great theoretical and practical potential.

6.4 The tree case for the resolution graph

If $\mathrm{rg}(F)$ is a tree, then we have polynomial-time SAT decision as follows:

1. First apply unit-clause-elimination, obtaining F' (where $\mathrm{rg}(F')$ still is a tree).
2. If $\bot \in F'$ then F' (and F) is unsatisfiable, otherwise F' (and F) is satisfiable, since repeated elimination of blocked clauses ([22, 23]) must now remove all clauses (the clause C labelling a leaf of $\mathrm{vig}(F')$ must be blocked, since only one literal in C is involved in a resolution while $|C| \geq 2$).

This gives a certain hope that SAT decision might be FPT in the treewidth of $\mathrm{rg}(F)$. (At this time, by using algorithm $\mathcal{TS}$ from Subsection 5.3 we can only show polynomial time SAT decision in the treewidth of the subsumption-resolution graph, given that the clause size is bounded by some fixed constant.)

7 Solving SAT problems by graph decomposition

7.1 Decomposable graph representations of (labelled) clause-sets

Specifying the generic requirements (I), (II) from Section 3 for the special case of SAT decision for (labelled boolean) clause-sets, we arrive at the following definition:

A graph representation $\mathcal{G}(F)$, which assigns to every labelled clause-set F a graph, is called **decomposable**, if the following data is associated with $\mathcal{G}$:

(1) There is an efficient SAT algorithm deciding satisfiability for "trivial" F, that is, where we have $|V(\mathcal{G}(F))| \leq 1$.

(2) In the case of $V(\mathcal{G}(F)) \neq \emptyset$ we can efficiently compute $F(G')$ for $G' \in \mathrm{cc}(\mathcal{G}(F))$ such that $\bigcup_{G' \in \mathrm{cc}(\mathcal{G}(F))} F(G') = F$ and $\mathcal{G}(F(G')) = G'$ for all $G' \in \mathrm{cc}(\mathcal{G}(F))$, and we have

$$F \in \mathcal{USAT} \Leftrightarrow \exists\, G' \in \mathrm{cc}(\mathcal{G}(F)) : F(G') \in \mathcal{USAT}. \tag{2}$$

(3) $\mathcal{G}(\varphi * F)$ is efficiently embeddable into $\mathcal{G}(F)$ via $j_{\varphi,F}$ for all partial assignments φ.

(4) For all $S \subseteq V(\mathcal{G}(F))$ we have an algorithm Φ computing a set $\Phi(F,S) \subseteq \mathcal{PASS}$ of partial assignments such that for all $\varphi \in \Phi(F,S)$ we have $S \cap j_{\varphi,F}(V(\mathcal{G}(\varphi * F))) = \emptyset$, and such that

$$F \in \mathcal{SAT} \Leftrightarrow \exists\, \varphi \in \Phi(F,S) : \varphi * F \in \mathcal{SAT}$$

holds.

Remarks:

1. Condition (1) makes sure that Step 1 in the generic algorithm $\mathcal{GS}(P)$ from Section 3 can be performed.
2. Condition (2) is the natural translation of requirement (I).
3. Conditions (3), (4) specialise requirement (II), where the "branches" now are realised by partial assignment (which eliminate the vertices from the separating vertex set S).
4. By the results in Subsection 6.1 it is $\mathrm{vig}(F)$ (and $\mathrm{cvg}(F)$) a decomposable graph representation.[14)]
5. By the results in Subsection 6.2 the presentations $\mathrm{cg}(F)$, $\mathrm{rg}(F)$ and $\mathrm{srg}(F)$ are decomposable graph representations.[15)]

So we can now apply the generic algorithms $\mathcal{GS}(F)$ from Section 3, or either its heuristic version $\mathcal{HS}(F)$ from Section 4 or the (tree decomposition based) version $\mathcal{TS}(F)$ from Subsection 5.3. It remains the problem of how to find good $\Phi(F,S)$ as required in condition (4).

7.2 How to branch efficiently?

When branching on *variables*, that is, S is a set of variables, then the canonical (first) choice for $\Phi(F,S)$ is the set of all $2^{|S|}$ partial assignments φ with $\mathrm{var}(\varphi) = S$. Very important in our setting now is to apply further processing to these φ, obtaining extensions $\varphi' \supseteq \varphi$, by enhancing them with forced assignments, at least adding unit clause propagations, and likely better exploiting the generalisation of unit clause propagation as investigated in [21, 25] (of course, if we have $\varphi' * F = \top$, then we found the formula satisfiable, while in case of $\bot \in \varphi' * F$ we can cancel this branch).

More complicated (and interesting) is the case of branching on *clauses*, that is, S is actually a clause-set. Here the canonical approach is to let $\Phi(F,S)$ be a "base for satisfying assignments", that is, a set of partial assignments φ which satisfy S, and such that for every partial assignment ψ satisfying S there exists some $\varphi \in \Phi(F,S)$ with $\varphi \subseteq \psi$. Note that the task of finding a base for satisfying assignments is exactly the same as finding a DNF equivalent to the CNF S, since $\Phi(F,S)$ is just a different encoding of a DNF.[16)] There are many algorithmic possibilities for such

[14)] Actually $\mathrm{vig}(F)$ does not precisely fulfil all the requirements, since $\mathrm{vig}(F)$ ignores empty clauses and unit clauses (while $\mathrm{cvg}(F)$ ignores variables which occur at most once); but since these special cases should be handled anyway in a special way, this is not really a problem.

[15)] Here again we do not precisely fulfil all the requirements due to the (small) problem with non-occurring variables, but again this is trivially handled.

[16)] Clauses in conjunctive normal forms directly state the falsifying assignments of the underlying boolean function, while clauses in disjunctive normal forms directly state the satisfying assignments of the underlying boolean function.

a conversion; most natural here seems to be to use a DPLL solver (without autarky reduction) which doesn't stop when finding a satisfying assignment (typically implementations for counting of satisfying assignments actually compute bases for satisfying assignments).

In general S might have only very large bases[17], and so here further processing as for branching on variables is essential. Actually, considering a case where S has many satisfying assignments, which almost all will be removed since they are inconsistent with F, it might be advantageous not to compute $\Phi(F, S)$ in two steps, but to consider a combined procedure, which directly outputs a "reduced base" of S w.r.t. F, consisting of satisfying partial assignments for S where we now only require that for every satisfying assignment ψ for F there must exist an element φ in this reduced base with $\varphi \subseteq \psi$.[18] For the computation of $\Phi(F, S)$ also the algorithms from [21, 25] based on the "hardness parameter" can be used; for clause-sets S of "hardness" k in time $O(n(S)^{2k})$ we can compute some $\Phi(F, S)$ of size at most $n(S)^k$.

To conclude, we would like to point out the special importance of the "afterburner" for our approach, when compared to the usual setting for exploiting tree decompositions: In order to achieve FPT, in Step 4 of the generic algorithm $\mathcal{GS}$ we can actually not afford to branch, but we somehow must have a general way of analysing all these branches together in one go; to obtain this "intelligence" seems not be possible in our general framework, however for compensation we can now afford (and should exploit(!)) that all these branches are very different (by exploiting the problem structure).

8 Experimental results

In this section we present the (very preliminary) experimental results we have gathered so far. At this early stage, we have only negative results, showing that just relying on tree decompositions is very likely infeasible for almost all instances and for any of the studied graph decomposition (this was expected, since the graph representations take only a very rough picture of the real problem).[19]

The real subject of this report, a tighter integration of (various) graph splitting methods and DPLL-like algorithms, couldn't be demonstrated experimentally yet in any convincing way due to (various) time constraints. We hope that within the next four months we will be doing good progress here.

We are using two levels of preprocessing:

1. reduction r_0: unit clause propagation, pure literals, subsumption
2. reduction r_1: r_0 and elimination of blocked clauses ([22, 23])

[17] and there is no bound on the number of elements of $\Phi(F, S)$ just depending on the number of clauses in F; if for example S consists just of one clause of size k, then $\Phi(F, S)$ needs to contain k elements; however we should point out that on the other hand a big S might have only very few satisfying assignments

[18] Of course, F should be taken into account only to a certain degree, since otherwise we would just solve the whole problem in this step.

[19] The resolution graphs take a closer look at the problem structure, but perhaps just because of the bigger graphs in almost all instances the treewidths obtained are worse than for the variable interaction graph; this could be due to a failure of the heuristics used to approximate the treewidth, but yet we do not have any indication for this (since correct data is hard to come by with).

It is quite easy to see that r_0 and r_1 are confluent (for stronger results see [20]), that is, the reduction result is uniquely determined.

Table 1: Data for various small benchmarks (`addm` and `hwb` from SAT03 competition; `bridge fault`, `dubois`, `phole`, `parity` and `pret` from `www.satlib.org`; and `lksat` and `unif` from SAT04 competition). Data for reduction r_1 is left out in case $r_0 = r_1$ on the benchmark.

benchmark	r	#var	#cls	$\|E(\text{vig})\|$	$\|E(\text{cvg})\|$	$\|E(\text{cg})\|$	$\|E(\text{rg})\|$	$\|E(\text{srg})\|$
addm-4-4	r_0	431	1451	1050	12738	7294	5436	5048
addm-4-4	r_1	431	1391	1050	12205	7004	5254	4866
addm-5-5	r_0	1074	3670	2643	32504	18601	13893	12885
addm-5-5	r_1	1074	3574	2643	31656	18140	13603	12595
bf1355-075	r_0	3301	2764	2422	46150	23358	19467	15089
bf1355-075	r_1	3301	1673	1719	28828	14769	12347	10639
bf1355-638	r_0	3385	2933	2763	50309	25451	21146	16185
bf1355-638	r_1	3385	2162	2439	37813	19508	16416	13437
dubois050	r_0	150	400	298	2968	1792	1184	1184
dubois100	r_0	300	800	598	5968	3592	2384	2384
hole08	r_0	72	297	540	2592	576	576	576
hole09	r_0	90	415	765	4050	810	810	810
hole10	r_0	110	561	1045	6050	1100	1100	1100
hwb-sat03.1607	r_0	134	628	505	15202	8598	6884	6496
hwb-sat03.1608	r_0	134	628	503	15046	8546	6804	6400
lksat-n900-936	r_0	872	3274	6389	58026	30010	29022	28920
lksat-n900-936	r_1	872	3216	6346	56468	29260	28340	28242
lksat-n900-937	r_0	871	3262	6391	58836	30221	29246	29154
lksat-n900-937	r_1	871	3198	6334	57052	29349	28447	28355
lksat-n900-938	r_0	869	3255	6371	58144	30044	29119	29003
lksat-n900-938	r_1	869	3193	6315	56383	29210	28361	28246
par16-1-c	r_0	317	1264	888	36128	19188	17221	16886
par16-1-c	r_1	317	1255	888	35997	19115	17176	16851
par16-2-c	r_0	349	1392	979	43584	23276	20853	20518
par16-2-c	r_1	349	1383	979	43453	23203	20808	20483
pret060-25	r_0	60	160	120	1200	720	480	480
pret150-25	r_0	150	400	300	3000	1800	1200	1200
unif-sat04-31	r_0	398	1691	4906	31926	16055	16013	16011
unif-sat04-32	r_0	400	1700	4943	32440	16259	16218	16214
unif-sat04-33	r_0	400	1700	4949	32602	16313	16276	16274

8.1 Treewidth

Computing the treewidth is very time consuming: Currently a good complete algorithm has worst case time complexity of $O(min(n^{k+2}, n^{n-k}))$ with n referring to the number of vertices and k to the treewidth ([13]). Hence, the exact treewidth could only be determined for some special structured graphs or graphs with fewer than say 100 vertices. However, since we are not interested in the exact treewidth,

but only in a "good" decomposition, approximation algorithms for the treewidth seem appropriate for this purpose.

In our experiments we used two approximation programs which we obtained from Arie Koster ([17]): `mdfi` and `gfi`. The first is based on the *Minimum Degree Fill-in* heuristic: Repeatedly a vertex of minimum degree is selected and removed from the graph to obtain a linear ordering. The second program is based on the *Greedy Fill-in* heuristic: A linear ordering of the vertices is constructed by repeatedly selecting a vertex that causes the least fill-in in the triangulation. The resulted ordering output from both programs were used to construct a tree decomposition. All experiments were performed on a system with an Intel 3.0 GHz CPU and 1 Gb of memory running on Fedora Core 4.

Table 3 shows for all five graphs upperbounds on the treewidth, while Table 4 shows the run-times. Two interesting observations can be made regarding this data. First, we know that for the exact treewidth

$$\mathrm{tw}(\mathrm{cvg}(F)) \geq \mathrm{tw}(\mathrm{cg}(F)) \geq \mathrm{tw}(\mathrm{rg}(F)) \geq \mathrm{tw}(\mathrm{srg}(F))$$

holds. However, the approximated treewidth computed by these algorithms does rarely show this linear order; actually on some benchmarks even a strict increment is shown. Apparently, removal of some edges can result quite often in a less accurate upperbound for the treewidth with these algorithms. The same phenomenon can also be observed for the effect of r_1 over r_0: We know that r_1 only removes clauses (and possibly also variables), and thus the treewidth cannot increase, however in quite a number of cases the effect of r_1 is, that the upper bound on the treewidth increases.

Second, tree-decompositions of variable-based graphs behave different than those of clause-based graphs. The most clear example of this observation are the `pigeon hole` instances: The treewidth of the clause-based graphs is linear in the number of pigeons, while the treewidth of the variable based graphs is quadratic in the number of pigeons. However, the "small" treewidth of clause-based graphs does not imply that these benchmarks can easily be solved using decomposition (if we are using splitting on clauses, then this can be very expensive).

On average, `gfi` appears to compute a more accurate upperbound on the treewidth than `mdfi`, but needs often substantially more memory and computational time.

The obvious conclusion from this data is, that just relying on the treewidth is likely to be a failure: There are some cases where the variable interaction graphs have appropriately small values, but in most cases also for this graph representation using treewidth-based algorithms as black boxes is useful only on very special instances.

9 Conclusion and future research

We presented a generic scheme $\mathcal{GS}$ for problem solving based on splitting graph representations, with two specialisations regarding the choice of splittings: A heuristical scheme $\mathcal{HS}$ using complexity estimations, and a scheme $\mathcal{TS}$ exploiting tree decompositions.

To apply these schemes for SAT solving, on the one hand we studied several graph representations, with the main novelty the resolution and subsumption-resolution graphs, and on the other hand we specified the branching means in the

notion of a "decomposable graph representation" of clause-sets. Bringing these two methods together, we discussed several techniques for efficient branching in this context.

Regarding complexity results, as a byproduct of our general framework we achieved polynomial time SAT decision in the treewidth of the subsumption-resolution graph for bounded clause-length. Finally we presented first experimental results on the effectiveness of the different graph representations.

This report should be seen as the starting point for a systematic exploration of the landscape of applications of graph splitting techniques to generalised SAT problems. From the many open problems mentioned, the following might be the most interesting ones (at this time):

1. Regarding general methods for finding splittings, the potential of hypergraph cuts (with all the existing optimisation options) needs to be explored. Another question is about planarity and the exploitation of these approaches.
2. How can FPT be achieved in our framework? We somehow believe that FPT should be possible for the resolution graph (also for unbounded clause-length).
3. Regarding the resolution graph, the connections between resolution refutations and the resolution graph on the one hand, and between satisfying assignments for the components of the resolution graph and satisfying assignments for the whole clause-sets on the other hand need to be understood.
4. Another important topic is a better understanding of the structure of separating edge sets in the resolution graph.
5. The algorithmic possibilities for branching on clauses need to be explored.
6. Regarding implementations, a natural starting point is to implement $\mathcal{HS}$ for various decomposable graph representations and various splitting schemes.
7. Can we exploit stronger preprocessing than given by r_0 and r_1 ?
8. Although the number of edges could be reduced substantially in several cases by preprocessing, and the number of edges in the resolution graph can be substantially smaller than the number of edges in the conflict graph or the common-variable graph, the heuristics used to approximate the treewidth seem to react to these improvements in a random fashion, sometimes yielding a better, sometimes a worse treewidth?!
9. And finally, much more extensive experimentation is needed, so that also a better understanding of the landscape of problem structures is achieved.

References

[1] Michael Alekhnovich and Alexander A. Razborov. Satisfiability, branch-width and Tseitin tautologies. In *Proceedings of the 43rd IEEE FOCS*, pages 593–603, 2002.

[2] Claude Berge. *Hypergraphs: Combinatorics of Finite Sets*, volume 45 of *North-Holland Mathematical Library*. North Holland, Amsterdam, 1989. ISBN 0 444 87489 5; QA166.23.B4813 1989.

[3] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:191–198, 2006.

[4] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In Giunchiglia and Tacchella [12], pages 315–329. ISBN 3-540-20851-8.

[5] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.

[6] Elizabeth Broering and Satyanarayana V. Lokam. Width-based algorithms for SAT and CIRCUIT-SAT. In Giunchiglia and Tacchella [12], pages 162–171. ISBN 3-540-20851-8.

[7] Pierre Duchet. Hypergraphs. In Ronald L. Graham, Martin Grötschel, and László Lovász, editors, *Handbook of Combinatorics, Volume 1*, chapter 7, pages 381–432. North-Holland, Amsterdam, 1995. ISBN 0-444-82346-8; QA164.H33 1995.

[8] Andrea Ferrara, Guoqiang Pan, and Moshe Y. Vardi. Treewidth in verification: Local vs. global. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2005)*, pages 489–503, 2005.

[9] Nicola Galesi and Oliver Kullmann. Polynomial time SAT decision, hypergraph transversals and the hermitian rank. In Holger H. Hoos and David G. Mitchell, editors, *The Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 76–85, Vancouver, British Columbia, Canada, May 2004.

[10] Nicola Galesi and Oliver Kullmann. Polynomial time SAT decision, hypergraph transversals and the hermitian rank. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 89–104, Berlin, 2005. Springer.

[11] Allen Van Gelder and Tai Joon Park. Partitioning methods for satisfiability testing on large formulas. *Information and Computation*, 162(1):179–184, October 2000.

[12] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing 2003*, volume 2919 of *Lecture Notes in Computer Science*, Berlin, 2004. Springer. ISBN 3-540-20851-8.

[13] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence (UAI-04)*, pages 201–208, Arlington, Virginia, 2004. AUAI Press.

[14] Georg Gottlob, Nicola Leone, and Franceso Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394–399. Morgan Kaufmann, 1999.

[15] Georg Gottlob, Francesco Scarcello, and Martha Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.

[16] Jack E. Graver and Mark E. Watkins. *Combinatorics with Emphasis on the Theory of Graphs*, volume 54 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1977. ISBN 0-387-90245-7; QA164.G7.

[17] Illya V. Hicks, Arie M. C. A. Koster, and Elif Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. In J. Cole Smith, editor, *TutORials 2005*, INFORMS TutORials in Operations Research Series, chapter 1, pages 1–33. INFORMS Annual Meeting, New Orleans, 2005.

[18] H.B. Hunt and R.E. Stearns. Power indices and easier hard problems. *Mathematical Systems Theory*, 23:209–225, 1990.

[19] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005. ISBN 0-321-29535-8; QA76.9.A43K54.

[20] Oliver Kullmann. Obere und untere Schranken für die Komplexität von aussagenlogischen Resolutionsbeweisen und Klassen von SAT-Algorithmen. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, April 1992. (Upper and lower bounds for the complexity of propositional resolution proofs and classes of SAT algorithms (in German); Diplomarbeit am Fachbereich Mathematik).

[21] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.

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

[23] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1-3):149–176, 1999.

[24] Oliver Kullmann. The combinatorics of conflicts between clauses. In Giunchiglia and Tacchella [12], pages 426–440. ISBN 3-540-20851-8.

[25] Oliver Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):303–352, March 2004.

[26] Oliver Kullmann and Horst Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages; the ps-file can be obtained at `http://cs-svr1.swan.ac.uk/~csoliver`, January 1997.

[27] Oliver Kullmann and Horst Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages; the ps-file can be obtained from `http://cs-svr1.swan.ac.uk/~csoliver`, December 1998.

[28] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[29] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, August 1980.

[30] Daniele Pretolani. Hypergraph reductions and satisfiability problems. In Giunchiglia and Tacchella [12], pages 383–397. ISBN 3-540-20851-8.

[31] Irina Rish and Rina Dechter. Resolution versus search: Two strategies for SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000 Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 215–259. IOS Press, Amsterdam, 2000. ISBN 1 58603 061 2.

[32] Marko Samer and Stefan Szeider. SAT and CSP of bounded treewidth revised. Preprint, March 2006.

[33] Alexander Schrijver. *Combinatorial Optimization*, volume A. Springer, Berlin, 2003. ISBN 3-540-44389-4; Series Algorithms and Combinatorics, no. 24.

[34] Navaratnasothie Selvakkumaran and George Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proceedings of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 726–733, 2003.

[35] Stefan Szeider. On fixed-parameter tractable parameterizations of SAT. In Giunchiglia and Tacchella [12], pages 188–202. ISBN 3-540-20851-8.

Table 2: Data for various large benchmarks (`connamacher`, `hoonsang/vis-bmc`, `ezfact`, and `kexu/frb` from SAT04 competition; `ferry` and `pyhala-braun` from SAT03 competition; and `longmult` from `www.cs.cmu.edu/~modelcheck/bmc/`). Data for reduction r_1 is left out in case $r_0 = r_1$ on the benchmark.

benchmark	r	#var	#cls	$\|E(\text{vig})\|$	$\|E(\text{cvg})\|$	$\|E(\text{cg})\|$	$\|E(\text{rg})\|$	$\|E(\text{srg})\|$
connm-sat04-941	r_0	1132	14011	15330	830448	41184	41184	41184
connm-sat04-944	r_0	1200	14892	16296	880416	43776	43776	43776
connm-sat04-947	r_0	1192	14506	15884	853776	42624	42624	42624
dekker-sat04-989	r_0	19183	57574	54115	2061130	1038870	997466	995946
dekker-sat04-989	r_1	19183	35380	49747	862617	430085	419927	418409
gigamax-sat04-984	r_0	9561	25554	24489	253274	130757	113920	112129
gigamax-sat04-984	r_1	9561	23573	23830	226372	117687	103157	102086
luckyS-sat04-990	r_0	8209	24063	23599	442899	225558	209272	207494
luckyS-sat04-990	r_1	8209	23076	23527	430409	219312	204110	202346
philo-sat04-988	r_0	12679	37946	35679	883718	447190	419815	418929
philo-sat04-988	r_1	12679	22508	33092	347202	181741	175756	175302
rcnum-sat04-987	r_0	25421	76360	70429	2508643	1265350	1209473	1204849
rcnum-sat04-987	r_1	25421	50627	65740	1067756	551416	533577	529344
ezfact-sat04-788	r_0	3008	19358	11631	931274	517911	436294	435581
ezfact-sat04-788	r_1	3008	19356	11631	931201	517868	436268	435567
ezfact-sat04-789	r_0	3007	19107	11428	1010079	557294	475792	474671
ezfact-sat04-789	r_1	3007	19107	11428	1010079	557294	475792	474671
ferry-sat03-378	r_0	2812	19478	27203	460922	116093	100060	99960
ferry-sat03-378	r_1	2812	16758	26436	354436	68290	66520	66464
ferry-sat03-379	r_0	2737	18946	26462	448173	112912	97318	97218
ferry-sat03-379	r_1	2737	16291	25695	344467	66377	64657	64601
frb65-sat04-873	r_0	780	24086	24021	1569495	43752	39462	39462
frb65-sat04-874	r_0	780	24066	24001	1560060	43712	39422	39422
longmult08	r_0	3552	8638	9520	312152	159857	152740	146566
longmult08	r_1	3552	7274	8253	222910	113794	108127	104407
longmult10	r_0	4566	11364	12461	396113	202881	193620	185904
longmult10	r_1	4566	10072	11260	306132	156422	148533	143409
longmult12	r_0	5660	14330	15638	483542	247681	236116	226850
longmult12	r_1	5660	13110	14503	393370	201114	190843	184291
pyhala-sat03-1543	r_0	7038	22221	17785	351475	186751	162490	158036
pyhala-sat03-1543	r_1	7038	22090	17785	345791	183924	159794	155347
pyhala-sat03-1544	r_0	7038	22221	17785	351475	186751	162490	158036
pyhala-sat03-1544	r_1	7038	22090	17785	345791	183924	159794	155347

Table 3: Upperbounds of the treewidth for various small benchmarks (`addm` and `hwb` from SAT03 competition; `bridge fault`, `dubois`, `phole`, `parity` and `pret` from `www.satlib.org`; and `lksat` and `unif` from SAT04 competition). Data for reduction r_1 is left out in case $r_0 = r_1$ on the benchmark. For each benchmark the best upperbound for the variable-based graph and the best upperbound for the clause-based graphs is shown in bold.

		vig(F)		cvg(F)		cg(F)		rg(F)		srg(F)	
benchmark	r	mdfi	gfi	mdfi	gfi	mdfi	gfi	mdfi	gfi	mdfi	gfi
addm-4-4	r_0	46	46	164	**159**	202	161	207	175	210	170
addm-4-4	r_1	46	**45**	186	168	202	162	196	172	225	166
addm-5-5	r_0	94	**76**	351	-	390	-	412	-	**335**	-
addm-5-5	r_1	94	**76**	351	-	390	-	370	-	**335**	-
bf1355-075	r_0	35	31	282	245	357	226	386	244	261	207
bf1355-075	r_1	40	**30**	277	215	266	207	254	191	277	**169**
bf1355-638	r_0	53	**45**	313	248	391	255	374	250	278	186
bf1355-638	r_1	54	49	260	218	285	210	249	188	258	**181**
dubois050	r_0	3	3	11	11	11	**9**	11	10	11	10
dubois100	r_0	3	3	11	11	11	**9**	11	10	11	10
hole8	r_0	**45**	**45**	32	32	**8**	**8**	**8**	**8**	**8**	**8**
hole9	r_0	**57**	**57**	39	37	**9**	**9**	**9**	**9**	**9**	**9**
hole10	r_0	71	**69**	47	46	**10**	**10**	**10**	**10**	**10**	**10**
hwb-as.sat03-1607	r_0	**30**	34	249	**209**	316	258	286	234	306	**209**
hwb-as.sat03-1608	r_0	**30**	31	265	231	279	272	280	232	302	**229**
lksat-n900-936	r_0	449	**442**	2159	-	1745	-	1741	-	1749	-
lksat-n900-936	r_1	452	443	2154	-	**1681**	-	1704	-	1690	-
lksat-n900-937	r_0	436	438	2134	-	1702	-	1705	-	1705	-
lksat-n900-937	r_1	436	**433**	2147	-	1680	-	**1660**	-	1665	-
lksat-n900-938	r_0	**436**	-	2176	-	1719	-	1728	-	1711	-
lksat-n900-938	r_1	**436**	442	2150	-	**1681**	-	1696	-	1682	-
par16-1-c	r_0	23	**19**	195	167	423	288	191	284	191	283
par16-1-c	r_1	23	**19**	195	**163**	423	286	191	293	191	283
par16-2-c	r_0	22	**21**	191	167	477	292	191	284	207	282
par16-2-c	r_1	22	**21**	191	**163**	477	-	191	284	207	281
pret60-25	r_0	9	**4**	**19**	**19**	23	**19**	23	**19**	23	**19**
pret150-25	r_0	**12**	**12**	**19**	**19**	23	23	27	27	27	27
unif-as.sat04-31	r_0	282	**276**	1284	-	**957**	-	969	-	969	-
unif-as.sat04-32	r_0	275	**273**	1325	-	**967**	-	**967**	-	**967**	-
unif-as.sat04-33	r_0	275	**270**	1305	-	969	-	**957**	-	**957**	-

Table 4: Time (s) to compute the upperbounds of the treewidth in table 3 for various small benchmarks (`addm` and `hwb` from SAT03 competition; `bridge fault`, `dubois`, `phole`, `parity` and `pret` from `www.satlib.org`; and `lksat` and `unif` from SAT04 competition). Data for reduction r_1 is left out in case $r_0 = r_1$ on the benchmark.

		vig(F)		cvg(F)		cg(F)		rg(F)		srg(F)	
benchmark	r	mdfi	gfi	mdfi	gfi	mdfi	gfi	mdfi	gfi	mdfi	gfi
addm-4-4	r_0	0.11	0.36	2.71	22.0	2.86	24.6	3.10	28.9	3.02	24.1
addm-4-4	r_1	0.11	0.36	2.81	25.1	2.95	25.2	2.86	27.5	2.96	26.3
addm-5-5	r_0	0.64	2.33	18.5	-	19.3	-	21.68	-	17.9	-
addm-5-5	r_1	0.65	2.33	18.5	-	19.3	-	20.8	-	17.93	-
bf1355-075	r_0	0.12	0.24	8.66	61.0	8.11	85.9	8.17	56.8	4.46	34.2
bf1355-075	r_1	0.10	0.21	6.94	43.3	4.89	44.0	4.50	30.4	4.15	23.6
bf1355-638	r_0	0.18	0.48	10.1	81.9	11.2	71.3	9.85	138	5.20	33.5
bf1355-638	r_1	0.17	0.46	7.66	48.3	6.54	55.0	5.28	41.4	4.83	35.5
dubois050	r_0	0.01	0.01	0.04	0.09	0.05	0.08	0.05	0.07	0.05	0.09
dubois100	r_0	0.01	0.02	0.15	0.23	0.20	0.23	0.17	0.24	0.17	0.25
hole8	r_0	0.02	0.09	0.06	0.49	0.02	0.05	0.02	0.05	0.02	0.05
hole9	r_0	0.03	0.19	0.15	0.93	0.04	0.10	0.03	0.09	0.04	0.08
hole10	r_0	0.06	0.34	0.29	1.82	0.06	0.15	0.05	0.16	0.05	0.15
hwb-as.sat03-1607	r_0	0.02	0.09	1.98	35.9	2.75	102	2.74	74.1	2.42	42.2
hwb-as.sat03-1608	r_0	0.02	0.07	2.36	28.7	2.50	42.2	2.46	39.3	2.53	37.9
lksat-n900-936	r_0	6.22	112	608	-	293	-	290	-	287	-
lksat-n900-936	r_1	6.37	113	592	-	269	-	272	-	268	-
lksat-n900-937	r_0	6.04	103	603	-	288	-	278	-	273	-
lksat-n900-937	r_1	6.10	104	586	-	277	-	263	-	259	-
lksat-n900-938	r_0	6.27	112	618	-	282	-	305	-	286	-
lksat-n900-938	r_1	6.16	113	599	-	265	-	267	-	268	-
par16-1-c	r_0	0.04	0.11	3.62	95.6	9.10	268	3.60	239	3.42	202
par16-1-c	r_1	0.04	0.10	3.54	96.5	8.87	263	3.57	536	3.47	234
par16-2-c	r_0	0.04	0.10	3.27	95.5	7.67	203	3.46	211	3.43	256
par16-2-c	r_1	0.04	0.12	3.27	101	7.83	308	3.44	161	3.37	253
pret60-25	r_0	0.00	0.00	0.02	0.06	0.02	0.06	0.02	0.05	0.02	0.05
pret150-25	r_0	0.01	0.01	0.07	0.07	0.08	0.08	0.08	0.08	0.08	0.08
unif-as.sat04-31	r_0	1.64	43.5	133	-	52.6	-	50.8	-	52.0	-
unif-as.sat04-32	r_0	1.56	50.5	138	-	51.7	-	50.8	-	52.4	-
unif-as.sat04-33	r_0	1.53	48.1	137	-	52.7	-	51.4	-	52.2	-