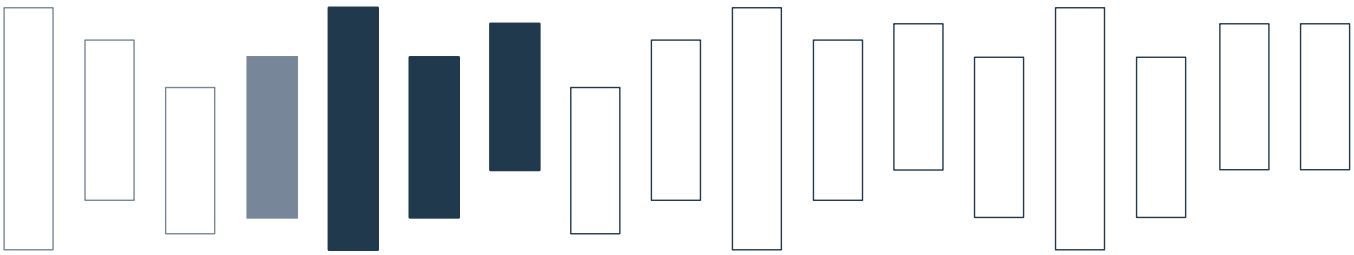
Technical Report AC-TR-20-005

June 2020

Threshold Treewidth and Hypertree Width

Robert Ganian, André Schidler,
Manuel Sorge, and Stefan Szeider



This is the authors' copy of a paper that will appear in the proceedings of IJCAI-PRICAI'20, the 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence.

www.ac.tuwien.ac.at/tr

Threshold Treewidth and Hypertree Width

Robert Ganian[1] , **André Schidler**[1] , **Manuel Sorge**[2] and **Stefan Szeider**[1]
[1]TU Wien, Vienna, Austria
[2]University of Warsaw, Warsaw, Poland
rganian@gmail.com, {aschidler,sz}@ac.tuwien.ac.at, manuel.sorge@mimuw.edu.pl

Abstract

Treewidth and hypertree width have proven to be highly successful structural parameters in the context of the Constraint Satisfaction Problem (CSP). When either of these parameters is bounded by a constant, then CSP becomes solvable in polynomial time. However, here the order of the polynomial in the running time depends on the width, and this is known to be unavoidable; therefore, the problem is not fixed-parameter tractable parameterized by either of these width measures. Here we introduce an enhancement of tree and hypertree width through a novel notion of thresholds, allowing the associated decompositions to take into account information about the computational costs associated with solving the given CSP instance. Aside from introducing these notions, we obtain efficient theoretical as well as empirical algorithms for computing threshold treewidth and hypertree width and show that these parameters give rise to fixed-parameter algorithms for CSP as well as other, more general problems. We complement our theoretical results with experimental evaluations in terms of heuristics as well as exact methods based on SAT/SMT encodings.

1 Introduction

The utilization of structural properties of problem instances is a key approach to tractability of otherwise intractable problems such as Constraint Satisfaction, Sum-of-Products, and other hard problems that arise in AI applications [Dechter, 1999; Gottlob *et al.*, 2002a]. The idea is to represent the instance by a (hyper)graph and to exploit its decomposability to guide dynamic programming methods for solving the problem. This way, one can give runtime guarantees in terms of the decomposition width. The most successful width measures for graphs and hypergraphs are treewidth and hypertree width, respectively [Gottlob *et al.*, 2014].

Treewidth. The Constraint Satisfaction Problem (CSP) can be solved in time $d^k \cdot n^{\mathcal{O}(1)}$ for instances whose primal graph has n vertices, treewidth k, and whose the variables range over a domain of size d [Dechter, 1999; Freuder, 1982]. If d is a constant, then this running time gives rise to fixed-parameter tractability w.r.t. the parameter treewidth [Gottlob *et al.*, 2002b]. However, without such a constant bound on the domain size, it is known that CSP it $\mathsf{W}[1]$-hard [Samer and Szeider, 2010] and hence not fixed-parameter tractable.

In the first part of this paper, we propose a new framework that allows fixed-parameter tractability even if some variables range over large domains. The idea is to exploit tree decompositions with the special property that each decomposition bag contains only a few (say, at most c) such high-domain variables whose domain size exceeds a given threshold d. This results in a new parameter for CSP that we call the threshold-d load-c treewidth. We show that finding such tree decompositions is fixed-parameter approximable, employing a replacement method which allows us to utilize state-of-the-art algorithms for computing treewidth such as Bodlaender *et al.*'s approximation [2016]. We then show that for any fixed c and d, CSP parameterized by threshold-d load-c treewidth is fixed-parameter tractable, and that the same tractability result can be lifted to other highly versatile problems such as the Sum-of-Products problem [Dechter, 1999; Koller and Friedman, 2009], Valued CSP [Schiex *et al.*, 1995; Živný, 2012], and the Integer Programming (IP) problem [Schrijver, 1986].

Hypertree width. Bounding the treewidth of a CSP instance automatically bounds the arity of its constraints. More general structural restrictions that admit large-arity constraints can be formulated in terms of the hypertree width of the constraint hypergraph. It is known that for any constant k, hypertree decompositions of width at most k can be found in polynomial time, and that CSP instances of hypertree width k can be solved in polynomial time. If k is a parameter and not constant, then both problems become $\mathsf{W}[1]$-hard and hence not fixed-parameter tractable. We show that also in the context of hypertree width, a more fine-grained parameter, which we call *threshold-d load-c hypertree width*, can be used to achieve fixed-parameter tractability. Here we distinguish between heavy and light hyperedges, where a hyperedge is light if the corresponding constraint is defined by a constraint relation that contains at most d tuples. Each bag of a threshold-d load-c hypertree decomposition of width k must admit an edge cover that consists of at most k hyperedges, where at most c of them are heavy. We show that for any fixed c and k, we can determine for a given hypergraph in polynomial time whether it admits a hypertree decomposition of width k where

the cover for each bag consists of at most c heavy hyperedges[1]. We further show that for any fixed c and d, given a width-k threshold-d load-c hypertree decomposition of a CSP instance, checking its satisfiability is fixed-parameter tractable when parameterized by the width k.

Practical algorithms and experiments. The most popular practical algorithms for finding treewidth and hypertree decompositions are based on characterizations in terms of *elimination orderings*. We show how these characterizations can be extended to capture threshold treewidth and threshold hypertree width. These then allow us to obtain practical algorithms that we test on large sets of graphs and hypergraphs originating from real-world applications. In particular, we propose and test several variants of the well-known min-degree heuristics, as well as exact methods based on SMT-encodings for computing threshold tree and hypertree decompositions. Our experimental findings are significant, as they show that by optimizing decompositions towards low load values we can obtain in many cases decompositions that are expected to perform much better in the dynamic programming phase than ordinary decompositions that are oblivious to the weight of vertices or hyperedges.

Related work. There are several reports on approaches for tuning greedy treewidth heuristics to improve the performance of particular dynamic programming (DP) algorithms. For instance, Kask *et al.* [2011] optimized the state space of graphical models for probabilistic reasoning, which corresponds in our setting to minimizing the product of the domain sizes of variables that appear together in a bag. Similar heuristics were suggested by Bachoore and Bodlaender [2007] for treewidth. Abseher *et al.* [2017] optimized heuristic tree decompositions w.r.t. the sizes of DP tables when solving individual combinatorial problems such as 3-Colorability or Minimum Dominating Set. Scarcello *et al.* [2007] presented a general framework for minimizing the weight of hypertree decompositions of bounded width. We discuss in Sections 3 and 4 how the above notions give rise to complexity parameters for CSP and how they compare to threshold treewidth and hypertree width.

2 Preliminaries

For an integer i, we let $[i] = \{1, 2, \ldots, i\}$ and $[i]_0 = [i] \cup \{0\}$. We let $\mathbb{N}$ be the set of natural numbers, and $\mathbb{N}_0$ the set $\mathbb{N} \cup \{0\}$. We refer to Diestel [2012] for standard graph terminology.

Similarly to graphs, a *hypergraph* H is a pair (V, E) where V or $V(H)$ is its vertex set and E or $E(H) \subseteq 2^V$ is its set of hyperedges. An *edge cover* of $S \subseteq V$ (in the hypergraph (V, E)) is a set $F \subseteq E$ such that for every $v \in S$ there is some $e \in F$ with $v \in e$. The *size* of an edge cover is its cardinality. For a (hyper)graph G, we will sometimes use $V(G)$ to denote its vertex set and $E(G)$ to denote the set of its (hyper)edges.

Parameterized complexity. In parameterized algorithmics [Downey and Fellows, 2013; Niedermeier, 2006], the running-time of an algorithm is studied with respect to a parameter $k \in \mathbb{N}_0$ and input size n. The basic idea is to find a parameter that describes the structure of the instance such that the combinatorial explosion can be confined to this parameter. In this respect, the most favorable complexity class is FPT (*fixed-parameter tractable*), which contains all problems that can be decided by an algorithm running in time $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a computable function. Algorithms with this running-time are called *fixed-parameter algorithms*.

Treewidth. A *tree decomposition* $\mathcal{T}$ of a (hyper)graph G is a pair (T, χ), where T is a tree and χ is a function that assigns each tree node t a set $\chi(t) \subseteq V(G)$ of vertices such that the following conditions hold:

(P1) For every (hyper)edge $e \in E(G)$ there is a tree node t such that $e \subseteq \chi(t)$.

(P2) For every vertex $v \in V(G)$, the set of tree nodes t with $v \in \chi(t)$ induces a non-empty subtree of T.

The sets $\chi(t)$ are called *bags* of the decomposition $\mathcal{T}$, and $\chi(t)$ is the bag associated with the tree node t. If G is a graph, then the *width* of a tree decomposition (T, χ) is the size of a largest bag minus 1. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum width over all tree decompositions of G.

Hypertree width. A *generalized hypertree decomposition* of a hypergraph H is a triple $\mathcal{D} = (T, \chi, \lambda)$ where (T, χ) is a tree decomposition of H and λ is function mapping each $t \in V(T)$ to an edge cover $\lambda(t) \subseteq E(H)$ of $\chi(t)$. The *width* of $\mathcal{D}$ is the size of a largest edge cover $\lambda(t)$ over all $t \in V(T)$, and the generalized hypertree width *ghtw*(H) of H is the smallest width over all generalized hypertree decompositions of H.

It is known to be NP-hard to decide whether a given hypergraph has generalized hypertree width ≤ 2 [Fischl *et al.*, 2018]. To make the recognition of hypergraphs of bounded width tractable, one needs to strengthen the definition of generalized hypertree width by adding a further restriction. A *hypertree decomposition* [Gottlob *et al.*, 2002a] of H is a generalized hypertree decomposition $\mathcal{D} = (T, \chi, \lambda)$ of H where T is a rooted tree that satisfies in addition to P1–P2 also the following Special Condition (P3):

(P3) If $t, t' \in V(T)$ are nodes such that t' is a descendant of t, then for each $e \in \lambda(t)$ we have $(e \setminus \chi(t)) \cap \chi(t') = \emptyset$.

The *hypertree width* *htw*(H) of H is the smallest width over all hypertree decompositions of H.

The constraint satisfaction problem. An instance of a *constraint satisfaction problem* (CSP) $\mathcal{I}$ is a triple (V, D, C) consisting of a finite set V of variables, a function D which maps each variable $v \in V$ to a set (called the *domain* of v), and a set C of constraints. A *constraint* $c \in C$ consists of a *scope* $S(c)$, which is a completely ordered subset of V, and a relation $R(c)$, which is a $|S(c)|$-ary relation on $\mathbb{N}$. The *size* of an instance $\mathcal{I}$ is $|\mathcal{I}| = |V| + |D| + \sum_{c \in C} |S(c)| \cdot |R(c)|$.

An *assignment* is a mapping $\theta : V \to \mathbb{N}$ which maps each variable $v \in V$ to an element of $D(v)$; a partial assignment is defined analogously, but for $V' \subseteq V$. A constraint $c \in C$ with scope $S(c) = (v_1, \ldots, v_{|S(c)|})$ is satisfied by a partial assignment θ if $R(c)$ contains the tuple $\theta(S(c)) = (\theta(v_1), \ldots, \theta(v_{|S(c)|}))$. An assignment is a *solution* if it satisfies all constraints in $\mathcal{I}$. The task in CSP is to decide whether the instance $\mathcal{I}$ has at least one solution.

The *primal graph* $G_{\mathcal{I}}$ of a CSP instance $\mathcal{I} = (V, D, C)$ is the graph whose vertex set is V and where two vertices v, w are adjacent if and only if there exists a constraint whose

[1] This is not fixed-parameter tractable for parameter k, as already without the c restriction, the problem is W[2]-hard.

scope contains both v and w. The *hypergraph* $H_{\mathcal{I}}$ of $\mathcal{I}$ is the hypergraph with vertex set V, where there is a hyperedge $E \subseteq V$ if and only if there exists a constraint with scope E.

3 Threshold Treewidth

The aim of this section is to define threshold treewidth for CSP, but to do that we first need to introduce a refinement of treewidth on graphs. Let G be a graph where V is bipartitioned into a set of light vertices and a set of heavy vertices; we call such graphs *loaded*. For $c \in \mathbb{N}_0$, a *load-c tree decomposition* of G is a tree decomposition of G such that each bag $\chi(t)$ contains at most c heavy vertices. It is worth noting that, while every graph admits a tree decomposition, for each fixed c there are loaded graphs which do not admit any load-c tree decomposition (consider, e.g., a complete graph on $c+1$ heavy vertices). The *load-c treewidth* of G is the minimum width of a load-c tree decomposition of G or ∞ if no such decomposition exists.

Let $d, c \in \mathbb{N}_0$ and $\mathcal{I} = (V, D, C)$ be a CSP instance. Moreover, let $G^d_{\mathcal{I}}$ be the primal graph such that $v \in V$ is light if and only if $|D(v)| \leq d$. Then the *threshold-d load-c treewidth* of $\mathcal{I}$ is the load-c treewidth of $G^d_{\mathcal{I}}$. The following theorem summarizes the key advantage of using the threshold-d load-c treewidth instead of the "standard" treewidth of $G_{\mathcal{I}}$.

Theorem 1. *Given $d, c \in \mathbb{N}$, a* CSP *instance $\mathcal{I}$ and a load-c tree decomposition of $G^d_{\mathcal{I}}$ of width k, it is possible to solve $\mathcal{I}$ in time at most $\mathcal{O}(d^k \cdot |\mathcal{I}|^{c+2})$.*

Proof Sketch. The proof follows by applying the classical algorithm for solving CSP by using the treewidth of the primal graph $G_{\mathcal{I}}$ [Freuder, 1982; Gottlob *et al.*, 2002b]. The stated runtime follows from the bound on high-domain variables imposed by the definition of load-c treewidth, which implies a bound of $d^k \cdot |\mathcal{I}|^c$ on the number of records stored for each node in the dynamic program. □

Bachoore and Bodlaender [2007] studied weighted treewidth with the motivation to measure the table size of dynamic programs directly as a weight of the decomposition. However, the weighted treewidth implicitly upper-bounds the domains of *all* variables. This is not the case for load-c treewidth, which allows each bag to contain up to c variables of arbitrarily large domains. Thus, load-c treewidth can be thought of as a more general parameter (fixed-parameter algorithms for it apply to a larger set of instances).

To apply Theorem 1 it is necessary to be able to compute a load-c tree decomposition of a loaded graph efficiently. While there is a significant body of literature on computing or approximating optimal-width tree decompositions of a given graph, it is not obvious at all how to directly enforce a bound on the number of heavy vertices per bag in any of the known state-of-the-art algorithms for the problem. Our next aim is to show that in spite of this, it is possible to reduce the problem of computing an approximate load-c tree decomposition to the problem of computing an optimal-width tree decomposition of a graph. This then allows us to use known results in order to find a sufficiently good approximation of load-c treewidth.

Lemma 1. *Given an n-vertex loaded graph G with m edges and an integer $k \geq 1$, it is possible to compute in $\mathcal{O}((n+m) \cdot k^2)$ time a graph G' such that:* (1) *If G has load-c treewidth k then G' has treewidth at most $ck+k$, and* (2) *given a tree decomposition of width ℓ of G', in linear time we can compute a load-$(\ell/(k+1))$ tree decomposition of G of width ℓ.*

Proof Sketch. Consider the graph G' constructed as follows: (a) we add each light vertex in G into G'; (b) for each heavy vertex $v \in V(G)$, we add $k+1$ vertices $v_0, v_1, \ldots, v_k$ into G' (we call them *images* of v); (c) we add an edge between each pair of images, say $v_i, v_j \in V(G')$, of some vertex v; (d) for each $vw \in E(G)$, we add into G' the edge vw (if both v and w are light), or the edges $\{ vw_i : i \in [k]_0 \}$ (if w was heavy and v was light), or the edges $\{ v_i w_j : i, j \in [k]_0 \}$ (if both v and w were heavy). Now it is not hard to show that every tree decomposition of G can be turned into an appropriate tree decomposition for G', and vice-versa. □

Lemma 1 and the algorithm of Bodlaender [1996] can be used to approximate load-c treewidth:

Theorem 2. *Given $c \in \mathbb{N}$, a loaded graph G and $k \in \mathbb{N}$, in $(ck)^{\mathcal{O}((ck)^3)} \cdot |V(G)|$ time it is possible to either correctly determine that the load-c treewidth of G is at least $k+1$ or to output a $(ck+k)$-width load-c tree decomposition of G.*

By constructing the graph $G^d_{\mathcal{I}}$ and then computing a load-c tree decomposition of $G^d_{\mathcal{I}}$ with width at most $ck+k$ using Theorem 2, in combination with Theorem 1, we obtain:

Theorem 3. *Given $c, d \in \mathbb{N}$, and a* CSP *instance $\mathcal{I}$, we can solve $\mathcal{I}$ in $d^{\mathcal{O}(ck)} \cdot |\mathcal{I}|^{c+2} + (ck)^{\mathcal{O}(ck)^3} \cdot |\mathcal{I}|^2$ time where k is the threshold-d load-c treewidth of $\mathcal{I}$. Thus, for constant c and d,* CSP *is fixed-parameter tractable parameterized by k.*

Further applications. While our exposition here focuses primarily on applications for the classical constraint satisfaction problem, it is worth noting that load-c treewidth can be applied analogously on many other prominent problems that arise in the AI context. By modifying the algorithm of Theorem 1 slightly, we can solve the following problems:

SUM-OF-PRODUCTS (SUMPROD) [Dechter, 1999]: Instead of ordinary constraints, we are given a set of weighted constraints; a *weighted constraint* assigns to each tuple in $R(c) \cup \{\bot\}$ a real number (where $\bot$ represents the case when c is not satisfied). A weighted constraint c naturally assigns a real number $c(\alpha)$ to each assignment α. The goal is to compute $\sum_\alpha \prod_{c \in C} c(\alpha)$ where the sum is taken over all assignments α for the given instance. This generalizes #CSP.

VALUED CSP (VCSP) [Schiex *et al.*, 1995; Živný, 2012]: Like in SUMPROD, we are given weighted constraints instead of ordinary ones, but now the goal is to find an assignment α minimizing $\sum_{c \in C} c(\alpha)$. This generalizes MAXCSP.

INTEGER PROGRAMMING (IP) [Schrijver, 1986]: The generalization of the famous INTEGER LINEAR PROGRAMMING problem to arbitrary polynomials. Herein, the primal graph is defined via co-occurrence of variables in polynomial constraints. The resulting loaded graph based on domain threshold can be defined in a similar way as for CSP. Differently from above, we may adapt a dynamic programming algorithm from the literature to use threshold-d load-c tree decompositions [Eiben *et al.*, 2019].

Proposition 1. *Given $c, d \in \mathbb{N}$, and an instance $\mathcal{I}$ of* SUMPROD*,* VCSP*, or* IP *we can solve $\mathcal{I}$ in time $d^{\mathcal{O}(ck)} \cdot |\mathcal{I}|^{c+2} + (ck)^{\mathcal{O}(ck)^3} \cdot |\mathcal{I}|^2$ where k is the threshold-d load-c treewidth of $\mathcal{I}$. Thus, for constant c, d,* SUMPROD*,* VCSP*, and* IP *are fixed-parameter tractable parameterized by k.*

4 Threshold Hypertree Width

In this section, we define threshold hypertree width for CSP, show how to use it to obtain fixed-parameter algorithms, and how to compute the associated decompositions. Similar to threshold treewidth, we will first introduce an enhancement of hypertree width for hypergraphs. Intuitively, the running time of dynamic programs for CSP based on decompositions of the corresponding hypergraph is strongly influenced by constraints, corresponding to hyperedges, whose relations contain many tuples. We hence aim to distinguish these hyperedges.

Let H be a hypergraph where $E = E(H)$ is bipartitioned into a set E_B of *light* hyperedges and a set E_R of *heavy* hyperedges. We call such hypergraphs *loaded*. Let $c \in \mathbb{N}_0$. A *load-c hypertree decomposition* of H is a hypertree decomposition (T, χ, λ) for H such that each edge cover $\lambda(v)$, $v \in V(T)$, contains at most c heavy hyperedges. The width and the notion of *load-c hypertree width* (of H) are defined in the same way as for hypertree decompositions.

Similar to threshold treewidth, for each fixed c there are hypergraphs that do not admit a load-c hypertree decomposition. For example, consider a clique graph with at least $c + 2$ vertices with heavy edges only, interpreted as a hypergraph.

We now apply the above notions to CSP. Let $d, c \in \mathbb{N}_0$ and $\mathcal{I} = (V, D, C)$ be a CSP instance. Let $H^d_{\mathcal{I}}$ be the loaded hypergraph of $\mathcal{I}$ wherein a hyperedge $F \in E(H^d_{\mathcal{I}})$ is light if and only if $|R(\gamma)| \leq d$, for the constraint $\gamma \in C$ corresponding to F, i.e., $S(\gamma) = F$. Then, the *threshold-d load-c hypertree width* of $\mathcal{I}$ is the load-c hypertree width of $H^d_{\mathcal{I}}$. For threshold-d load-c hypertree width, we also obtain a fixed-parameter algorithm for CSP. Instead of building on hypertree decompositions in the above, we may also use generalized hypertree decompositions, leading to the notion of *generalized threshold-d load- hypertree width* and the associated decompositions.

Theorem 4. *Given $c, d \in \mathbb{N}$, a* CSP *instance $\mathcal{I}$ of (generalized) threshold-d load-c hypertree width, and the associated decomposition of $H^d_{\mathcal{I}}$, in $\mathcal{O}(d^k \cdot |\mathcal{I}|^{c+2})$ time it is possible to decide $\mathcal{I}$ and produce a solution if there is one.*

In particular, for fixed c, d, CSP is fixed-parameter tractable parameterized by k when a threshold-d load-c hypertree decomposition of width k is given.

Proof Sketch. A usual approach used for ordinary hypertree decompositions is to compute an equivalent CSP whose hypergraph is acyclic and then use an algorithm for acyclic CSPs [Gottlob *et al.*, 2002a]. We instead apply a direct dynamic programming approach; the stated running-time bound then follows from the upper bound on constraints with large number of tuples imposed by the definition of load-c hypertree width.

Let (T, χ, λ) be the load-c hypertree decomposition of $H^d_{\mathcal{I}}$ provided in the input. Root T arbitrarily and denote the root by r. Define V_t to be the set of vertices in bag t and all bags below t. A *t-mapping* is a mapping that assigns to each variable $v \in \chi(t)$ a value from $D(v)$.

The algorithm proceeds by dynamic programming, i.e., computing, for each node $t \in V(T)$ in a leaf-to-root fashion, the set $M(t)$ of all t-mappings θ with the following two properties: (1), there exists some extension θ' of θ to V_t which maps each variable $v \in V_t$ to an element of $D(v)$ such that each constraint γ with $S(\gamma) \subseteq V_t$ is satisfied by θ' and, (2), for each constraint $\gamma \in \lambda(t)$, mapping θ projected[2] onto $S(\gamma)$ occurs as a tuple in γ projected onto $\chi(t)$.

It is not hard to prove that $\mathcal{I}$ is a YES-instance if and only if $M(r) \neq \emptyset$; the solution can then be reconstructed by retracing the dynamic program in a standard fashion. We omit the details of computing $M(t)$ and the correctness and running time proofs. The main idea for the running time is the fact that, due to property (2), each t-mapping $\theta \in M(t)$ arises from combining tuples of the relations of constraints in the cover $\lambda(t)$ and hence $|M(t)| \leq d^{k-c} \cdot |\mathcal{I}|^c$. □

Similar to weighted treewidth, a weighted variant of hypertreewidth has been proposed [Scarcello *et al.*, 2007] wherein the whole decomposition (T, χ, λ) is weighted according to the estimated running time of running a dynamic program similar to the above. The approach is, slightly simplified, to weigh each hyperedge in the cover of a bag by $|R(c)|$ for the corresponding constraint c and then to minimize $\sum_{t \in V(T)} \Pi_{c \in \lambda(t)} |R(c)|$. A drawback here again is that, using this quantity as a parameter, it implicitly bounds the number of tuples in each constraint $|R(c)|$ and in turn all domain sizes. This is not the case for threshold-d load-c hypertree width.

We now turn to computing the decomposition for the hypergraph of the CSP used in Theorem 4. A previous approach for computing ordinary hypertree decompositions of width at most k [Gottlob *et al.*, 1999] can be modified to work also for load-c hypertree decomposition of width at most k. Indeed, this approach has previously been adapted within an even more general framework [Scarcello *et al.*, 2007], which allows to compute hypertree decompositions of width at most k that additionally optimize a weight function from a specific class of functions. Applying this framework leads to the following.

Theorem 5. *Given $c, k \in \mathbb{N}$, and a loaded hypergraph H, in $\mathcal{O}(|E(H)|^{2k} \cdot |V(H)|^2)$ time it is possible to compute a load-c hypertree decomposition for H or correctly report that no such decomposition exists.*

Assuming FPT $\neq$ W[2] the running time in Theorem 5 cannot be improved to a fixed-parameter tractable one, even if c is constant. This follows from the fact that the special case of deciding whether a given hypergraph without heavy hyperedges admits a load-0 hypertree decomposition of width at most k is W[2]-hard with respect to k [Gottlob *et al.*, 2005].

Bounding the threshold treewidth or threshold hypertree width of a CSP instance constitutes a hybrid restriction and not a structural restriction [Carbonnel and Cooper, 2016], as these restrictions are formulated in terms of the loaded primal graphs and the loaded hypergraphs, and not in terms of the

[2]The projection of a relation R onto a subset S of its variables is the set resulting from taking each tuple of R and removing from this tuple the entries for variables not in S.

plain, unlabeled (hyper)graphs. However, as the loaded (hyper)graphs carry only very little additional information, we would like to label such restrictions as *semi-structural*.

5 Elimination Orderings and Algorithms

Our algorithms rely on a different characterization of treewidth and generalized hypertree width, which we describe below.

An *elimination ordering* $\prec$ of a graph G is a total ordering $\prec$ of $V(G)$. Let us denote the i-th vertex in $\prec$ as v_i, and let $G_0 = G$. For $i \in [|V|]$, let the graph G_i be obtained from G_{i-1} by removing v_i and adding edges between each pair of vertices in the neighborhood of v_i (i.e., the neighborhood, $N_{G_{i-1}}(v_i)$, of v_i in G_{i-1} becomes a clique). The *width* of v_i w.r.t. $\prec$ is then defined as $|N_{G_{i-1}}(v_i)|$, and the *width* of $\prec$ is the maximum width over all vertices in G w.r.t. $\prec$. It is well known that width-k elimination orderings precisely correspond to tree decompositions of width k [Kloks, 1994; Bodlaender and Koster, 2010]. Generalized hypertree decompositions also have a corresponding elimination ordering characterization: the only difference is that instead of the width of v_i, we consider its cover-width, which is the size of a minimum edge cover of $|N_{G_{i-1}}(v_i)|$ in $H_{\mathcal{I}}$ [Fichte *et al.*, 2018].

It is relatively straightforward to adapt these notions of elimination orderings to describe threshold treewidth and threshold generalized hypertree width. In particular, one can show:

Theorem 6. (1) *A* CSP *instance $\mathcal{I}$ has threshold-d load-c treewidth k if and only if $G^d_{\mathcal{I}}$ admits an elimination ordering of width k with the property that for each v_i, $N_{G^d_{\mathcal{I},i-1}}(v_i)$ contains at most c heavy vertices.* (2) *A* CSP *instance $\mathcal{I}$ has generalized threshold-d c-hypertree width k if and only if $G^d_{\mathcal{I}}$ admits an elimination ordering of cover width k with the property that for each v_i, $N_{G^d_{\mathcal{I},i-1}}(v_i)$ admits a hyperedge cover (in $H_{\mathcal{I}}$) of size at most k containing at most c heavy hyperedges.*

A (significantly more complicated) elimination ordering characterization of hypertree width has been obtained by Schidler and Szeider [2020]. This, too, can be translated into a characterization of threshold-d load-c hypertree width. However, experimental evaluations confirmed the expectation that there was no practical benefit to using hypertree width instead of generalized hypertree width.

Algorithms. An optimal elimination ordering without taking heavy vertices into account for a given graph can be computed using a SAT encoding [Samer and Veith, 2009]; below we call this algorithm `TW-X-Obl`. This encoding can be extended to compute optimal generalized hypertree decompositions, by computing the necessary covers [Fichte *et al.*, 2018] using an SMT encoding, below denoted by `HT-X-Obl`. We also call these methods (load) *oblivious*. The SMT approach is highly robust and can be adapted to also compute threshold-d load-c tree decompositions: analogously to the existing cardinality constraints for bags/covers, we add new constraints that limit the number of heavy vertices/hyperedges (see Theorem 6). We use this to either aim for a decomposition of optimal width and secondarily with minimum load (leading to algorithms `TW-X-W→L` and `HT-X-W→L`), or one of optimal load and secondarily with minimum width (leading to algorithms `TW-X-L→W` and `HT-X-L→W`).

Since optimal elimination orderings of graphs are hard to compute, heuristics are often used. The *Min-Degree* method constructs an ordering in a greedy fashion by choosing v_i among the vertices of minimal degree in G_{i-1}, and yields good treewidth values overall [Bodlaender and Koster, 2011]; below we call this algorithm `TW-H-Obl` and say it is *oblivious*. We adapted this method into two new heuristics that consider load: `TW-H-L→W` and `TW-H-W→L`. The former chooses all the heavy vertices first, which leads to decompositions with low loads but possibly larger width. The latter maintains a bound ℓ on the target load of the decomposition, and selects a vertex v_i of minimum degree among those with the property that they contain at most ℓ heavy neighbors in G_{i-1}; if no such vertex exists, the heuristic restarts with an incremented value of ℓ.

Our heuristics for generalized hypertree width begin by computing an elimination ordering for the primal graph using min-degree [Dermaku *et al.*, 2008]. The *oblivious* heuristic now uses branch-and-bound to compute an optimal cover for each bag, called `HT-H-Obl` below. We also use two extensions for computing the covers while taking the load into account: one that optimizes for load first and width second (`HT-H-L→W`), and one that optimizes for width first and load second (`HT-H-W→L`). In all three heuristics, the computed covers are actually optimal for their respective objective.

6 Experiments

In this section we present experimental results using the algorithms discussed in the previous section. We were particularly interested in the difference in loads between oblivious (`Obl`) and width-first load-second (`W→L`) methods, and the trade-off between width-first (`W→L`) and load-first (`L→W`) methods.

Setup. We ran our experiments on a cluster, where each node consists of two Xeon E5-2640 CPUs, each running 10 cores at 2.4 GHz and 160 GB memory. As solvers we used *minisat 2.2.0* (http://minisat.se/) and *optimathsat 1.6.2* (http://optimathsat.disi.unitn.it/). The control code and heuristics use *Python 3.8.0*. The nodes run *Ubuntu 18.04*. We used a 8 GB memory limit and a 2 hour time limit per instance.

Plots. Most of our plots use a specific type of scatterplot. The position of the marker shows the pairs of values of the data point, while the size of the marker shows the number of instances for which these values were obtained. The measured quantities are noted in the plot caption. For example, the data points in Figure 1a are, for each of the solved instances, the pair of loads of the tree decompositions computed by the `TW-X-W→L` and `TW-X-Obl` methods from Section 5.

Instances. For threshold-d load-c treewidth we used 2788 instances from the *twlib*[3] benchmark set. For generalized threshold-d load-c treewidth we used the 3071 hyperbench[4] instances after removing self-loops and subsumed hyperedges. We created our loaded instances by marking a certain percentage of all vertices or hyperedges as heavy. We ran experiments for different ratios, but since the outcomes did not deviate too much, here we only present the results for a ratio of 30% heavy vertices/hyperedges (same as by Kask *et al.* [2011]).

[3] http://www.staff.science.uu.nl/~bodla101/treewidthlib/

[4] http://hyperbench.dbai.tuwien.ac.at/

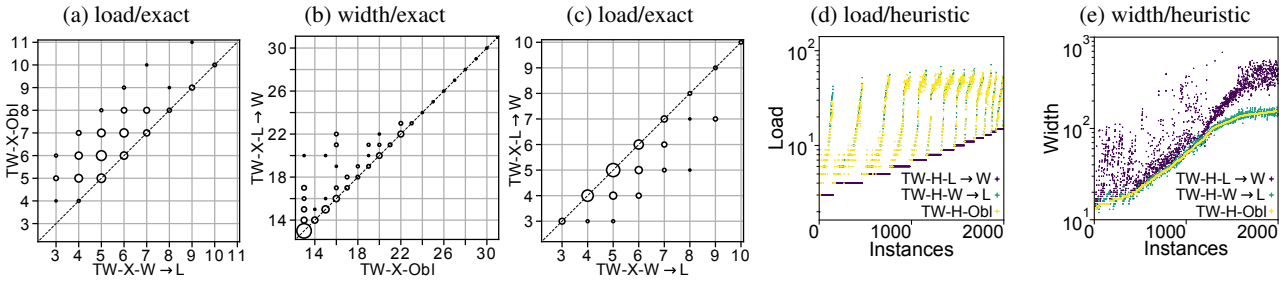


Figure 1: Exact and heuristic treewidth computations: differences in values depending on the optimization strategy.

Since instances of low width are considered efficiently solvable, our presentation only focuses on high-width instances. In particular, for treewidth and generalized hypertree width, we disregarded instances of width below 13 and below 4, respectively. We were not able to find solutions for all instances and for this reason state the number of instances in the results.

Treewidth. Figures 1a to 1c show the results from running the exact algorithms (methods `TW-X`; 168 instances could be solved within the time limit). It shows that even by using `W→L` methods, we can significantly improve the load without increasing the treewidth. Further improvements in load can be obtained by using `TW-X-L→W`, as seen in Figure 1c. In Figure 1b we see that the trade-off (in terms of the treewidth) required to achieve the optimal loads is often very small.

The results are different for heuristic methods. Figures 1d and 1e show the results from the 2203 instances with high width. While good estimates for load or width are possible, finding good estimates for both at the same time is not possible with the discussed heuristics: In Figure 1d we see that both the `TW-H-Obl` and `TW-H-W→L` heuristics mostly fail to find a good estimate for the load. On the other hand, Figure 1e shows that `TW-H-L→W` tends to result in decompositions with much higher treewidth than the optimum. These results suggest that it may be non-trivial to obtain heuristics which provide a good trade-off between load and width.

Generalized hypertree width. Figures 2a to 2c show the results from 259 optimal decompositions computed within the time limit. The general outlook is the same as for treewidth: Even the `HT-X-W→L` algorithm significantly improves the load without any trade-off (Figure 2a), and `HT-X-L→W` can decrease the load even further (Figure 2a) while only slightly increasing the generalized hypertree width (Figure 2c).

The results obtained by applying the `HT-H-Obl` and `HT-H-L→W` heuristics on the 1624 instances with large width can be seen in Figure 2d. There is a stark contrast to the heuristics used for treewidth: The `HT-H-W→L` heuristic can significantly reduce the load with no trade-off, as the width is guaranteed to be the same (i.e. fixed after giving the ordering).

7 Concluding Remarks

We have introduced a novel way of refining treewidth and hypertree width via the notion of thresholds, allowing us to lift previous fixed-parameter tractability results for CSP and other problems beyond the reach of classical width parameters. Our new parameters have the advantage over the standard

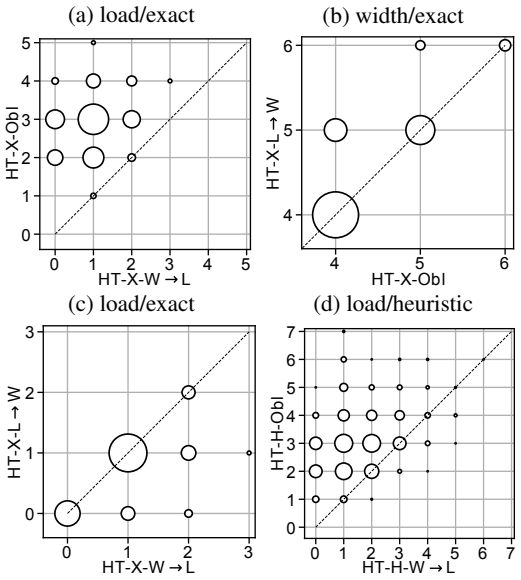


Figure 2: Exact and heuristic generalized hypertreewidth computations: differences in values depending on the optimization strategy.

"oblivious" variants of treewidth and hypertree width that they can take more instance-specific information into account. A further advantage of our new parameters is that decompositions that optimize our refined parameter can be used as the input to existing standard dynamic programming algorithms, resulting in a potential exponential speedup. Our empirical findings show that in realistic scenarios, one can expect that optimizing the loads requires only minimal overhead while offering huge gains in further processing times.

Acknowledgments

André Schidler and Stefan Szeider acknowledge the support from the FWF, projects P32441 and W1255, and from the WWTF, project ICT19-065. Robert Ganian also acknowledges support from the FWF, notably from project P31336. Manuel Sorge acknowledges support by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement no. 714704.

References

[Abseher *et al.*, 2017] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. *JAIR*, 58:829–858, 2017.

[Bachoore and Bodlaender, 2007] Emgad Bachoore and Hans L. Bodlaender. Weighted treewidth — algorithmic techniques and results. In *ISAAC'07*, volume 4835 of *LNCS*, pages 893–903, 2007.

[Bodlaender and Koster, 2010] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations i. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[Bodlaender and Koster, 2011] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. lower bounds. *Inf. Comput.*, 209(7):1103–1119, 2011.

[Bodlaender *et al.*, 2016] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016.

[Bodlaender, 1996] H. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[Carbonnel and Cooper, 2016] Clément Carbonnel and Martin C. Cooper. Tractability in constraint satisfaction problems: a survey. *Constraints*, 21(2):115–144, 2016.

[Dechter, 1999] Rina Dechter. Bucket elimination: a unifying framework for reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.

[Dermaku *et al.*, 2008] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *MICAI 2008: Advances in Artificial Intelligence*, pages 1–11. Springer Berlin Heidelberg, 2008.

[Diestel, 2012] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173. Springer, 2012.

[Downey and Fellows, 2013] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[Eiben *et al.*, 2019] Eduard Eiben, Robert Ganian, Dusan Knop, and Sebastian Ordyniak. Solving integer quadratic programming via explicit and structural restrictions. In *AAAI'19*, pages 1477–1484, 2019.

[Fichte *et al.*, 2018] Johannes K. Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In *CP'18*, volume 11008 of *LNCS*, pages 109–127, 2018.

[Fischl *et al.*, 2018] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and fractional hypertree decompositions: Hard and easy cases. In *PODS 2018*, pages 17–32, 2018.

[Freuder, 1982] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *J. of the ACM*, 29(1):24–32, 1982.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On tractable queries and constraints. In *DEXA'99*, volume 1677 of *LNCS*, pages 1–15, 1999.

[Gottlob *et al.*, 2002a] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comp. Sys. Sci.*, 64(3):579–627, 2002.

[Gottlob *et al.*, 2002b] Georg Gottlob, Francesco Scarcello, and Martha Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artif. Intell.*, 138(1-2):55–86, 2002.

[Gottlob *et al.*, 2005] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG'05*, volume 3787 of *LNCS*, pages 1–15, 2005.

[Gottlob *et al.*, 2014] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. In *Tractability: Practical Approaches to Hard Problems*, pages 3–38. Cambridge University Press, 2014.

[Kask *et al.*, 2011] Kalev Kask, Andrew Gelfand, Lars Otten, and Rina Dechter. Pushing the power of stochastic greedy ordering schemes for inference in graphical models. In *AAAI-11*, 2011.

[Kloks, 1994] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[Koller and Friedman, 2009] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. MIT Press, 2009.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and Its Applications. OUP Oxford, 2006.

[Samer and Szeider, 2010] Marko Samer and Stefan Szeider. Constraint satisfaction with bounded treewidth revisited. *J. Comp. Sys. Sci.*, 76(2):103–114, 2010.

[Samer and Veith, 2009] Marko Samer and Helmut Veith. Encoding treewidth into SAT. In *SAT 2009*, pages 45–50, 2009.

[Scarcello *et al.*, 2007] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. *J. Comp. Sys. Sci.*, 73(3):475–506, 2007.

[Schidler and Szeider, 2020] André Schidler and Stefan Szeider. Computing optimal hypertree decompositions. In *ALENEX20*, pages 1–11, 2020.

[Schiex *et al.*, 1995] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *IJCAI-95*, pages 631–639, 1995.

[Schrijver, 1986] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[Živný, 2012] Stanislav Živný. *The Complexity of Valued Constraint Satisfaction Problems*. Springer, 2012.