Journal of Artificial Intelligence Research 74 (2022) 1687-1713 Submitted 01/2022; published 08/2022

Threshold Treewidth and Hypertree Width

Robert Ganian RGANIAN@AC.TUWIEN.AC.AT
André Schidler ASCHIDLER@AC.TUWIEN.AC.AT
Manuel Sorge MANUEL.SORGE@AC.TUWIEN.AC.AT
Stefan Szeider SZ@AC.TUWIEN.AC.AT
Algorithms and Complexity Group
Faculty of Informatics
TU Wien
Vienna, Austria

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, Pichler, & Wei, 2010; Gottlob, Leone, & Scarcello, 2002). 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, Greco, & Scarcello, 2014).

1.1 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 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, Scarcello, & Sideri,

2002). However, without such a constant bound on the domain size, it is known that CSP is W[1]-hard (Samer & 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 (though finite) 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 approximable to within a factor of $(c+1)$ in fixed-parameter time, employing a replacement method which allows us to utilize state-of-the-art algorithms for computing treewidth such as Bodlaender *et al.*'s approximation (Bodlaender, Drange, Dregi, Fomin, Lokshtanov, & Pilipczuk, 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 CSP with Default Values (Ganian, Kim, Slivovsky, & Szeider, 2018, 2022), Valued CSP (Schiex, Fargier, & Verfaillie, 1995; Zivny, 2012), and the Integer Programming (IP) problem (Schrijver, 1999).

1.2 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 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.

1.3 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

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

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.

1.4 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, Jégou and Terrioux (2017) considered computing tree decompositions whose bags induce connected subgraphs in order to speed up solution methods whose running time depends on the connected components induced by bags. Kask, Gelfand, Otten, and Dechter (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, Musliu, and Woltran (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, Greco, and Leone (2007) presented a general framework for minimizing the weight of hypertree decompositions of bounded width. We discuss in Sections 3 and 5 how the above notions give rise to complexity parameters for CSP and how they compare to threshold treewidth and hypertree width.

1.5 Outline

We give the basic definitions and notation in Section 2. In Section 3 we formally introduce the notion of threshold-d load-c treewidth and give results on computing the associated decompositions. In Section 4 we give applications of these new notions to further prominent problems different from CSP in the AI context. In Section 5 we then introduce threshold-d load-c hypertree width and give results on computing the associated decompositions. In Section 6 we give alternative characterizations of the threshold treewidth and hypertree width notions via so-called elimination orderings which we use in our experiments. The algorithms we implemented are described in Section 7 and in Section 8 we report on the empirical results. Section 9 contains a conclusion.

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.

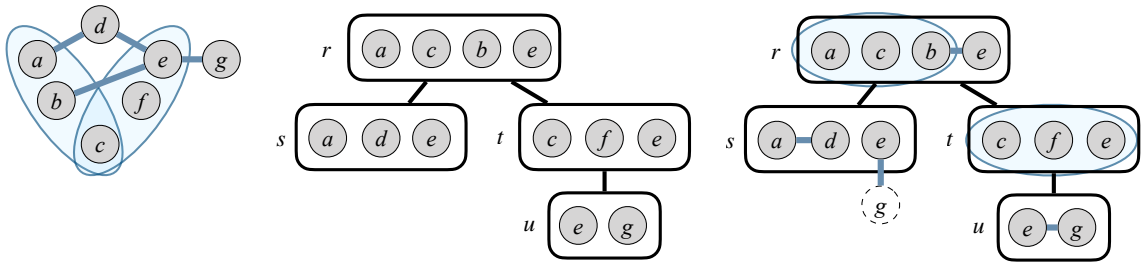

Figure 1: Left: a hypergraph H. Middle: a tree decomposition of H of width 3. Right: a hypertree decomposition of H of width 2; the covers of the bags are indicated by the blue edges and blue encircled vertex sets. Observe that the hypertree decomposition satisfies the Special Condition: the bag at node s is the only bag whose edge cover uses an edge containing a vertex from outside the bag (the edge $\{e, g\} \in \lambda(s)$ contains the vertex g outside $\chi(s)$). However, as s has no descendants, the Special Condition is trivially satisfied.

2.1 Parameterized Complexity

In parameterized algorithmics (Downey & Fellows, 2013; Niedermeier, 2006; Cygan, Fomin, Kowalik, Lokshtanov, Marx, Pilipczuk, Pilipczuk, & Saurabh, 2015; Flum & Grohe, 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*. A less favorable outcome is an XP *algorithm*, which is an algorithm running in time $\mathcal{O}(n^{f(k)})$; problems admitting such algorithms belong to the class XP. Problems hard for the complexity classes W[1], W[2], ..., W[P] do not admit fixed-parameter algorithms (even though they might be in XP) under standard complexity assumptions.

2.2 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. The *width* of a tree decomposition (T, χ) is the size of a largest bag minus 1. The *treewidth* of a (hyper)graph G, denoted by $\mathrm{tw}(G)$, is the minimum width over all tree decompositions of G.

2.3 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, Gottlob, & Pichler, 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., 2002) 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) and (P2) also the following Special Condition (P3):

(P3) If $t, t' \in V(T)$ are nodes in T such that t' is a descendant[2] 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.

To avoid trivial cases, we consider only hypergraphs $H = (V, E)$ where each $v \in V$ is contained in at least one $e \in E$. Consequently, every considered hypergraph H has an edge cover, and the parameters $ghtw(H)$ and $htw(H)$ are always defined. If $|V| = 1$ then $htw(H) = ghtw(H) = 1$.

Figure 1 shows a hypergraph, a tree decomposition, and a hypertree decomposition.

2.4 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*, denoted by $S(c)$, which is a completely ordered subset of V, and a relation, denoted by $R(c)$, which is a $|S(c)|$-ary relation on $\mathbb{N}$. If not stated otherwise, we assume that for each scope there is at most one constraint with that scope. 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 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. Note that the hypergraph does not contain parallel edges as for each scope there is at most one constraint with that scope.

2. A *descendant* of a node t in a tree T is any node t' on a path from t to a leaf of T in the subtree rooted at t.

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 $\mathcal{O}(d^{k+1} \cdot |\mathcal{I}|^{c+2})$.*

Proof. 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., 2002), whereas the stated runtime follows from the bound on high-domain variables imposed by the definition of load-c treewidth. However, since the proof idea is also used in the subsequent Propositions 1 to 3, we provide a full description of the algorithm below for completeness.

Let $\mathcal{T} = (T, \chi)$ be the load-c tree decomposition of $G^d_{\mathcal{I}}$ provided on the input. Choose an arbitrary node t of T and denote it as the root r. Let $V_t = \{\, v \in V : v \in \chi(t) \vee (\text{ there is a child } t' \text{ of } t \text{ such that } v \in \chi(t') \,)\}$. Moreover, let a *t-mapping* be a mapping that assigns to each variable v in $\chi(t)$ a value from $D(v)$. It is easy to see that the number of t-mappings is upper-bounded by $d^{k-c+1} \cdot |\mathcal{I}|^c$.

The algorithm proceeds by computing, for each node t in a leaf-to-root fashion, the set $M(t)$ of all t-mappings with the following property: $\theta \in M(t)$ if and only if there exists an extension θ' of θ to V_t such that each constraint q with $S(q) \subseteq V_t$ is satisfied by θ'. Clearly, $\mathcal{I}$ is a YES-instance if and only if $M(r)$ is non-empty; moreover, if we correctly compute a non-empty $M(r)$ by leaf-to-root dynamic programming, then it is possible to reconstruct a solution for $\mathcal{I}$ by retracing the steps of the dynamic program in a standard fashion.

To compute $M(\ell)$ for a leaf ℓ, it suffices to loop over all ℓ-mappings and for each perform a brute-force check to determine whether all of the relevant constraints are satisfied. For a non-leaf node t, we also loop over all t-mappings, whereas for each t-mapping θ we first check whether each constraint c such that $S(c) \subseteq \chi(t)$ is satisfied; if not, we discard θ. If yes, we then check that θ is "consistent" with each of the children of t—notably, for each child t' of t, we ensure that there is at least one t'-mapping θ' such that $\forall v \in \chi(t) \cap \chi(t') : \theta'(v) = \theta(v)$.[3] If this is the case then we add θ to $M(t)$.

3. This check can be carried out in amortized constant time via suitable data structures if all t-mappings are ordered based on a fixed variable ordering.

Correctness follows by the observation that each constraint c such that $S(c) \subseteq V_t$ must be contained in a bag of at least one descendant of t, and hence each such constraint is checked against θ by transitivity. The runtime bound follows by the upper bound on $V(T)$ and the upper bound on the number of t-mappings for each node t. □

We now briefly discuss the relation between threshold-d load-c treewidth and other parameters of CSP instances related to treewidth and domain size. First, Bachoore and Bodlaender (2007) introduced a parameter called weighted treewidth. Consider a graph G with vertex-weight function $w\colon V(G) \to \mathbb{N}$. The *weighted width* of a tree decomposition (T, χ) of G is $\max_{t \in V(T)} \Pi_{v \in \chi(t)} w(v)$, and the minimum such quantity is the *weighted treewidth* of G. The *weighted treewidth* of a CSP instance is the weighted treewidth of its primal graph with weight function w defined as $w(v) = |D(v)|$ for each variable v. It is not hard to see that we can replace the given load-c tree decomposition in Theorem 1 by a tree decomposition minimizing the weighted treewidth, say the minimum is w, and the algorithm would run in $\mathcal{O}(w \cdot |\mathcal{I}|^2)$ time. 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, that is, fixed-parameter algorithms for it apply to a larger set of instances.

Another way of dealing with variables with large domain would be to replace each of these variables v in every constraint by $\lceil \log |D(v)| \rceil$ *representative* variables with domain size two. Since the representative variables occur together in a constraint, they induce a clique in the primal graph. Computing a tree decomposition of low width for the primal graph thus roughly corresponds to minimizing the number of high-domain variables in a bag. More precisely, it corresponds to minimizing the sum of the logarithms of the domain sizes of the high-domain variables in the bags. Similarly to weighted treewidth, this means that the (maximum) domain size is in a strong relation with the width. In comparison, the approach taken here is aimed at restricting the number of high-domain variables that occur together in a bag.

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 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. 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).

Clearly, G' can be constructed from G in time $\mathcal{O}((n+m) \cdot k^2)$. For the part (1) of the lemma, consider a minimum-width tree decomposition $\mathcal{T} = (T, \chi)$ of G. Now consider the mapping χ' that is obtained from χ by replacing each occurrence of a heavy vertex v by all of its images, i.e., $v_0, \ldots, v_k$—formally, $x \in \chi'(t \in V(T))$ if and only if either $x \in \chi(t)$, or there exists $v \in V(G)$ such that $v \in \chi(t)$ and $x = v_i$. Since the number of heavy vertices in a single bag was upper-bounded by c, the maximum size of an image of χ' is $(k+1) \cdot c + k + 1 - c = ck + k + 1$. It is easy to verify that (T, χ') is a tree decomposition of G', and so the first claim follows.

For part (2) of the lemma, call a tree decomposition $\mathcal{T}' = (T, \chi')$ of G' *discrete* if for each $v \in V(G)$ such that v is heavy and each $t \in V(T)$ it holds that either for all $i \in [k]_0$ we have $v_i \notin \chi'(t)$ or for all $i \in [k]_0$ we have $v_i \in \chi'(t)$. Let $\widehat{\mathcal{T}} = (T, \widehat{\chi})$ be a tree decomposition of G' of width at most ℓ. We first claim that in linear time we can compute a tree decomposition $\mathcal{T}' = (T, \chi')$ of G' that is discrete and of width at most ℓ. To do this, we compute χ' from $\widehat{\chi}$ as follows. We iterate over all $t \in V(T)$ and for each vertex in $\widehat{\chi}(t)$ we check whether it is the image of some heavy vertex $v \in V(G)$ and, if so, we check whether all images of v are contained in $\widehat{\chi}(t)$. If not all images of v are contained in $\widehat{\chi}(t)$ we remove from $\widehat{\chi}(t)$ all images of v. In this way we obtain a mapping χ'. Note that, for each t, the above computation can be done in $\mathcal{O}(|\widehat{\chi}(t)|)$ time as follows. First, iterate over $\widehat{\chi}(t)$, obtaining a list of heavy vertices which have images in $\widehat{\chi}(t)$. For each such vertex v, initialize an empty list of images in $\widehat{\chi}(t)$. Iterate over $\widehat{\chi}(t)$ again to fill the lists of images with pointers to the images in $\widehat{\chi}(t)$. Finally, compute the length of each list and, if it is shorter than $k+1$, remove all images from $\widehat{\chi}(t)$ using the pointers. Thus, (T, χ') can be computed in linear time.

Next, we argue that (T, χ') is a tree decomposition of G'. Consider first condition (P2) of tree decompositions. Clearly, (P2) holds for every vertex v which is not an image of a heavy vertex. For the sake of contradiction, assume that (P2) is violated for an image v_j, $j \in [k]_0$, of some heavy vertex $v \in V(G)$. Thus, there are $r, s, t \in V(T)$ such that s is on the unique path between r and t in T, $v_j \in \chi'(r)$, $v_j \in \chi'(t)$, and $v_j \notin \chi'(s)$. Observe that both $\widehat{\chi}(r)$ and $\widehat{\chi}(t)$ contain all images of v whereas there is an image v_i of v which is not contained in $\widehat{\chi}(s)$. Hence, (P2) is violated for $(T, \widehat{\chi})$ and vertex v_i, a contradiction.

Now consider condition (P1). Clearly, (P1) holds for each edge whose endpoints either both are images of a heavy vertex of G or both are not images of heavy vertex of G. For the sake of contradiction, assume that (P1) does not hold for an edge such that one endpoint, u, is not the image of a heavy vertex and one endpoint, v_j for some $j \in [k]_0$, is the image of a heavy vertex $v \in V(G)$. Since the images of v induce a clique in G, there is a node $t \in V(T)$ such that $\widehat{\chi}(t)$ contains all images of v.[4] By assumption on u, we have $u \notin \chi'(t)$ and thus $u \notin \widehat{\chi}(t)$. There is thus an edge e in T whose removal separates T into a connected component that contains t and a connected component that contains all $t' \in T$ such that

4. This is a well-known fact about cliques and tree decompositions and can be proved roughly as follows: The vertices in the clique induce subtrees of the decomposition tree whose vertex sets have pairwise nonempty intersection. Since the trees are subtrees of the decomposition tree, this means there is a vertex in the decomposition tree that is contained in all of the subtrees.

$u \in \chi'(t')$. Moreover, there is such an edge e such that one endpoint, s, has the property that $u \in \chi'(s)$. Since u is adjacent to each $v_i \in V(G')$, $i \in [k]_0$, for each $i \in [k]_0$ there is $r_i \in V(T)$ such that both $u, v_i \in \widehat{\chi}(r_i)$. By (P2) of $(T, \widehat{\chi})$, for each $i \in [k]_0$, the subtree of T induced by the nodes $r \in V(T)$ with $v_i \in \widehat{\chi}(r)$ contains e. Thus, $\widehat{\chi}(s)$ contains each v_i. By construction of χ' it follows that $\chi'(s)$ contains each v_i. This is a contradiction to the fact that $\chi'(s)$ contains u and to the assumption that there is no bag of (T, χ') that contains both u and v_j. Thus, (P2) holds for (T, χ').

Above we have shown that the discrete tree decomposition (T, χ') of G' of width ℓ can be computed in linear time. Next, let us compute the mapping χ from χ' as follows: For each $t \in V(T)$, we put $v \in \chi(t)$ if either $v \in \chi'(t)$ or there exists $j \in [k]_0$ such that $v_j \in \chi'(t)$. Since in this way each vertex in a bag $\chi'(t)$ can only lead to the addition of at most one vertex into $\chi(t)$, it is easy to see that the maximum size of an image of χ is $\ell + 1$. Hence, if (T, χ) is a tree decomposition, then its width is at most ℓ.

We claim that the load of (T, χ) is at most $\ell/(k+1)$. Otherwise, there would be some $t \in V(T)$ such that $\chi(t)$ contains $\ell/(k+1) + 1$ heavy vertices. In that case, by discreteness of χ', the number of vertices in $\chi'(t)$ is at least $(k+1) \cdot (\ell/(k+1) + 1) = \ell + k + 1 > \ell + 1$. This contradicts the fact that (T, χ') has width ℓ.

It remains to show that (T, χ) is a tree decomposition of G. Condition (P1) clearly holds for every edge $vw \in E(G)$ such that $vw \in E(G')$. On the other hand, if $vw \notin E(G')$ then either one or both of v, w are heavy in G, and hence, e.g., the vertices v_0 and w_0 are adjacent in G'. This implies that there is some node $t' \in V(T)$ such that $\{v_0, w_0\} \subseteq \chi'(t')$, and by construction we obtain $\{v, w\} \subseteq \chi(t')$—hence (P1) holds. Finally, assume that (P2) is violated. Since it is easy to see that each vertex in $V(G)$ will be contained in at least one image of χ, this means that there would be some $v \in V(G)$ and nodes $t, t_a, t_b \in V(T)$ such that:

- $v \notin \chi(t)$ but $v \in \chi(t_a)$ and $v \in \chi(t_b)$;
- t separates t_a from t_b in T.

If v is light, then this would immediately violate the fact that $\mathcal{T}'$ is a tree decomposition of G'. On the other hand, if v is heavy, then there would have to exist v_i and v_j such that $v_i \in \chi'(t_a)$ and $v_j \in \chi'(t_b)$; moreover, $v_i \neq v_j$ since otherwise we would once again contradict (P2) for $\mathcal{T}'$. But then by construction we know that $v_i v_j \in E(G')$. Thus, by (P1) there is a bag $t' \in V(T)$ for which $v_i, v_j \in \chi'(t')$. By (P2) there is a path in T from t_a (resp. from t_b) to t' on which each bag s has $v_i \in \chi(s)$ (resp. $v_j \in \chi(s)$). One of these paths contains t and thus $v \in \chi(t)$, a contradiction. Hence (P2) holds as well, completing the proof. □

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 with $\mathcal{O}(|V(G)|)$ nodes.*

Proof. First, we construct the graph G' as per Lemma 1. By that lemma, if G has load-c treewidth at most k, then G' has treewidth at most $ck + k = \ell$. We then apply the fixed-parameter linear-time algorithm for treewidth of Bodlaender (1996) to compute a tree decomposition of width at most ℓ, or correctly determine that no such tree decomposition exists—in which case we output "NO". Applying this algorithm takes $\ell^{\mathcal{O}(\ell^3)} \cdot |V(G)|$ time

(see also Bodlaender et al. (2016)). If the output is NO, then the load-c treewidth of G is at least $k+1$, as required. If a decomposition for G' is found, we translate it back to G using Lemma 1 and output the result. By Lemma 1 the treewidth of the output decomposition is at most $ck+k$ and the load is at most

$$\frac{\ell}{k+1} = \frac{c(k+1)+k-c}{k+1} = c + \frac{k-c}{k+1}.$$

Since the load is an integer, it is at most c, as claimed. □

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 $d^{ck+k+1} \cdot |\mathcal{I}|^{c+2} + (ck)^{\mathcal{O}((ck)^3)} \cdot |\mathcal{I}|$ 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.*

Proof. The algorithm is as follows. We first construct the graph $G^d_{\mathcal{I}}$. Since $\mathcal{I}$ has threshold-d load-c treewidth at most k, the maximum number of variables in a constraint is at most $k+1$. Thus, $G^d_{\mathcal{I}}$ can be computed in $O(k^2 \cdot |\mathcal{I}|)$ time by initializing an empty graph with a vertex for each variable of $\mathcal{I}$, marking the vertices as heavy that correspond to variables with domain size more than d, and then iterating over all constraints and adding the corresponding edges. Then, we compute a load-c tree decomposition of $G^d_{\mathcal{I}}$ with width at most $ck+k$ using Theorem 2. This takes $ck^{\mathcal{O}((ck)^3)} \cdot k^2 \cdot |\mathcal{I}|$ time. The result then follows from Theorem 1. □

Note that the runtime bound stated in Theorem 3 would allow us to take the threshold d as an additional parameter instead of a constant, to still establish fixed-parameter tractability of CSP, parameterized by $k+d$.

4. Further Applications of Threshold Treewidth

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. In this subsection, we outline three such applications of our machinery in highly general settings.

4.1 Weighted Constraint Satisfaction with Default Values

Our first application concerns a recently introduced extension of constraint satisfaction via a combination of weights and default values (Brault-Baron, Capelli, & Mengel, 2015; Ganian et al., 2022); see also the published preprint (Ganian et al., 2018)). This extension captures, among others, counting CSP (#CSP) and counting SAT (#SAT). We introduce the extension below by building on our preliminaries on CSP.

For a variable set V and a domain D, a *weighted constraint C of arity ρ over D with default value η* (or "weighted constraint" in brief) is a tuple $C = (S, F, f, \eta)$ where

- the *scope* $S = (x_1, \ldots, x_\rho)$ is a sequence of variables from V,
- $\eta \in \mathbb{Q}$ is a rational number called the *default value*,

- $F \subseteq D^\rho$ is called the *support*, and
- $f : F \to \mathbb{Q}$ is a mapping which assigns rational weights to the support.

A weighted constraint $c = (S, F, f, \eta)$ naturally induces a total function on assignments of its scope $S = (x_1, \ldots, x_\rho)$: for each assignment $\alpha : X \to D$ where $X \supseteq S$, we define the *value* $c(\alpha)$ of c under α as $c(\alpha) = f(\alpha(x_1), \ldots, \alpha(x_\rho))$ if $(\alpha(x_1), \ldots, \alpha(x_\rho)) \in F$ and $c(\alpha) = \eta$ otherwise.

Similarly to CSP, an instance $\mathcal{I}$ of Weighted Constraint Satisfaction with Default Values (#CSPD) is a tuple (V, D, C), but here C is a set of weighted constraints. The task in #CSPD is to compute the total weight of all assignments of V, i.e., to compute the value $\mathsf{sol}(\mathcal{I}) = \sum_{\alpha:V\to D} \prod_{c\in C} c(\alpha)$.

#CSPD was shown to be fixed-parameter tractable when parameterized by the treewidth of the primal graph plus $|D|$ (Ganian et al., 2018), in particular as a corollary of a more general dynamic programming algorithm $\mathbb{A}$ (Ganian et al., 2018, Theorem 1). When $\mathbb{A}$ is applied on the primal graph, it proceeds in a leaf-to-root fashion that is similar in nature to the algorithm described in the proof of Theorem 1 here; however, formally the records stored by $\mathbb{A}$ are more elaborate. In particular, at each node t of a provided tree decomposition, $\mathbb{A}$ stores one record for each pair $(\theta, \vec{B})$ where

- θ is an assignment of the vertices in $\chi(t)$, and
- $\vec{B}$ is a tuple that specifies for each constraint that is "processed" at t the subset of tuples in the support that agree with θ.

Crucially, when applying $\mathbb{A}$ on the primal graph, in every tuple $(\theta, \vec{B})$ the latter component $\vec{B}$ is fully determined by the former component. And since the number of possible choices for θ is upper-bounded by $d^k \cdot |\mathcal{I}|^{c+2}$ for the same reason as in Theorem 1, we obtain:

Proposition 1. *Given $c, d \in \mathbb{N}$, and an instance $\mathcal{I}$ of* #CSPD *it is possible to solve $\mathcal{I}$ in $d^{ck+k+1} \cdot |\mathcal{I}|^{c+2} + (ck)^{\mathcal{O}((ck)^3)} \cdot |\mathcal{I}|$ time where k is the threshold-d load-c treewidth of $\mathcal{I}$. In particular, for constant c and d* #CSPD *is fixed-parameter tractable parameterized by k.*

4.2 Valued Constraint Satisfaction

The second application is for the Valued CSP (VCSP) (Schiex et al., 1995; Zivny, 2012). Herein, we are given the same input as in #CSPD but where every weighted constraint has a default value of 0. The goal in VCSP is to compute a variable assignment α that minimizes $\sum_{c\in C} c(\alpha)$. VCSP generalizes MaxCSP, where we aim to find an assignment for a CSP instance that maximizes the number of satisfied constraints.

It is a folklore result that VCSP can be solved by a dynamic programming algorithm along a tree decomposition of the primal graph, yielding XP-tractability when parameterized by the treewidth of the primal graph (Carbonnel, Romero, & Zivný, 2018; Bertele & Brioschi, 1972). The algorithm can be seen as a slight extension of the one presented in Theorem 1: the records $M(t)$ used in the algorithm that keep a list of all assignments θ are enhanced to also keep track of the value $\sum_{c\subseteq V_t} c(\theta)$. We thus obtain the following.

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

4.3 Integer Programming

Our third application concerns Integer Programming (IP) (Schrijver, 1999), the generalization of the famous Integer Linear Programming problem to arbitrary polynomials. IP is, in fact, *undecidable* in general; see Köppe (2012) for a survey on its complexity. However, when there are explicit bounds on the variable domains, it can be solved by a fixed-parameter algorithm via dynamic programming on tree decompositions.

For our presentation, we provide a streamlined definition of IP with domain bounds as used, e.g., by Eiben, Ganian, Knop, and Ordyniak (2019). An instance of IP consists of a tuple $(X, \mathcal{F}, \beta, \gamma)$ where:

- $X = \{x_1, \ldots, x_n\}$ is a set of variables,
- $\mathcal{F}$ is a set of integer polynomial inequalities over variables in X, that is, inequalities of the form $p(x_1, \ldots, x_n) \leq 0$ where p is a sparsely encoded polynomial with rational coefficients,
- β is a mapping from variables in X to their domain, i.e., $\beta(x)$ is the set of all integers z such that $x \mapsto z$ satisfies all constraints in $\mathcal{F}$ over precisely the variable x (these are often called *box constraints*), and
- γ is an integer polynomial over variables in X called the evaluation function.

The goal in IP is to find an assignment α of the variables of $\mathcal{I}$ which (1) satisfies all inequalities in $\mathcal{F}$ and β while achieving the maximum value of γ.

Let $d = \max_{x \in X} |\beta(x)|$, and let the primal graph $G_{\mathcal{I}}$ of an IP instance $\mathcal{I}$ be the graph whose vertex set is X and where two variables are adjacent if and only if there exists an inequality in $\mathcal{F}$ containing both variables. It is known that IP is fixed-parameter tractable when parameterized by d plus the treewidth of $G_{\mathcal{I}}$ (Eiben et al., 2019). The algorithm $\mathbb{B}$ used to establish this result performs leaf-to-root dynamic programming that is analogous in spirit to the procedure used in the proof of Theorem 1. Herein in particular, at each node t algorithm $\mathbb{B}$ stores records which specify the most favorable "partial evaluation" of γ for each possible assignment of variables in $\chi(t)$ in view of β and $\mathcal{F}$.

Since each variable is equipped with a domain via β, we may define the graph $G^d_{\mathcal{I}}$ in an analogous way as for CSP. Once that is done, it is not difficult to verify that running the algorithm of Eiben et al. (2019) on a threshold-d load-c tree decomposition of $G^d_{\mathcal{I}}$ guarantees a runtime bound for solving IP of $d^{\mathcal{O}(k)} \cdot |\mathcal{I}|^{c+2}$. In combination with our Theorem 2, we conclude:

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

5. 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. As a load-c hypertree decomposition contains a tree decomposition for the clique, there is a bag containing all vertices of this clique, and the minimum edge cover for this bag has size $c + 1$.

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-c hypertree width* and the associated decompositions.

Theorem 4. *Given $c, d \in \mathbb{N}$, a* CSP *instance $\mathcal{I}$ with (generalized) threshold-d load-c hypertree width k together with 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., 2002). 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. For each $t \in V(T)$, let $V_t = \bigcup_{t'} \chi(t')$, where the union is taken over all t' in the subtree of T rooted at 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[5] onto $S(\gamma)$ occurs as a tuple in γ projected onto $\chi(t)$.

Observe that $\mathcal{I}$ is a YES-instance if and only if $M(r) \neq \emptyset$: The backward direction follows from property (1). To see the forward direction, note that any satisfying assignment projected onto $\chi(r)$ is contained in $M(r)$. Thus, to decide $\mathcal{I}$ it suffices to compute all sets $M(t)$, $t \in V(T)$. The solution, if it exists, can then be computed by retracing the steps of the dynamic program in a standard fashion.

Before we explain how to compute $M(t)$, consider the following way of constructing a t-mapping θ. For each constraint in $\lambda(t)$, pick a tuple such that each pair of picked tuples agree on the variables they share (if any). Note that the picked tuples induce a t-mapping, and we set θ to be this mapping. Call a t-mapping constructed in this way *derived*. Note that the number of derived t-mappings is at most $d^{k-c} \cdot |\mathcal{I}|^c$ and that the set of all derived t-mappings can be computed in $\mathcal{O}(d^{k-c} \cdot |\mathcal{I}|^{c+1})$ time.

Next, we explain how to compute $M(t)$. To compute $M(\ell)$ for a leaf ℓ, due to property (2), it suffices to loop over all derived ℓ-mappings and to put them into $M(\ell)$ if they satisfy all constraints γ for which $S(\gamma) \subseteq \chi(\ell)$. By the bound on the number of derived ℓ-mappings, this takes $\mathcal{O}(d^k \cdot |\mathcal{I}|^{c+1})$ time.

Consider an internal node t of T. Again, we loop over each derived t-mapping θ and check whether it satisfies all constraints whose scope is in $\chi(t)$. If not, then we discard θ. If yes, then for each child t' of t we check whether there is a mapping $\theta' \in M(t')$ such that θ and θ' agree on their shared variables; in formulas $\forall v \in \chi(t) \cap \chi(t') : \theta'(v) = \theta(v)$. If so, then we put θ into $M(t)$. By using property (2) of the mappings in $M(t')$, in this way, we correctly compute $M(t)$. Using suitable data structures and the bound on the number of derived mappings, this computation can be carried out in time at most $\mathcal{O}(d^k \cdot |\mathcal{I}|^{c+1})$ per node in T. □

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 by first recursively decomposing the input hypergraph via separators which roughly correspond to the vertex sets of the potential covers of the bags, that is, sets S of at most k hyperedges. The decomposition can then be determined in a bottom-up fashion (Gottlob, Leone, & Scarcello, 1999). This approach can be adapted to load-c hypertree decomposition by replacing the sets S with sets of at most k hyperedges among which there are at most c heavy hyperedges. We omit the details. Indeed, we may instead use a more general framework, due to Scarcello et al. (2007), which allows to compute

5. 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.

hypertree decompositions of width at most k that additionally optimize an arbitrary weight function. 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 of width at most k or correctly report that no such decomposition exists.*

Proof. We first state the result of Scarcello et al. (2007) in a simplified and weaker form that is sufficient for our purpose. Let p be a function that assigns an integer to a bag of any hypertree decomposition. Let r_p be the function of the running time needed to evaluate p. A *tree aggregation function* is a function that assigns to each hypertree decomposition (T, χ, λ) the integer $\max_{t \in V(T)} p(t)$. Scarcello et al.'s Theorems 4.4 and 4.5 now imply the following. There is an algorithm that, given an integer k, a hypergraph H, and a tree aggregation function f, computes a width-k hypertree decomposition for H that minimizes f, or correctly decides that no such decomposition exists. The algorithm runs in $\mathcal{O}(|E(H)|^{2k} \cdot |V(H)| \cdot (|V(H)| + r_p))$ time.[6]

To apply this result to our setting, we put p to be the function that assigns to each bag t the number of heavy hyperedges in the edge cover $\lambda(t)$. Thus, Scarcello et al.'s algorithm will compute the smallest c such that there is a load-c hypertree decomposition. Note that $|\lambda(t)| \leq |V(H)|$ and hence $r_p = \mathcal{O}(|V(H)|)$. This implies the running-time bound. □

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, Grohe, Musliu, Samer, & Scarcello, 2005).

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

6. Elimination Orderings

The algorithms used in our experiments rely on a characterization of treewidth and generalized hypertree width by so-called elimination orders. 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 each $i \in [|V(G)|]$, 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 in G_i). 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 a graph G has treewidth k if and only if it admits an elimination ordering $\prec$ of width k (Kloks, 1994; Bodlaender & Koster, 2010). Moreover, a tree decomposition of width k can be computed from such $\prec$ and, vice-versa, given a tree decomposition

6. The running time bound follows from the analysis given by Scarcello et al. (2007) in the proof of Theorem 4.5.

of width k one can construct a width-k elimination ordering in polynomial time (Kloks, 1994; Bodlaender & Koster, 2010).

Recently, it has been shown that generalized hypertree decompositions of CSP instances can be characterized in a similar way (Fichte, Hecher, Lodha, & Szeider, 2018). In particular, consider a CSP instance $\mathcal{I}$ with primal graph $G_\mathcal{I}$ and an elimination ordering $\prec$ of $G_\mathcal{I}$. The *cover width* of v_i w.r.t. $\prec$ is then defined as the size of a minimum edge cover of $N_{G_{i-1}}(v_i) \cup \{v_i\}$ in $H_\mathcal{I}$, and the *cover width* of $\prec$ is the maximum cover width over all vertices in G w.r.t. $\prec$. Analogously as in the treewidth case, a generalized hypertree decomposition of width k can be computed from an elimination ordering $\prec$ of cover width k, and, vice-versa, given a generalized hypertree decomposition of width k one can construct a cover width-k elimination ordering in polynomial time (Fichte et al., 2018; Schidler & Szeider, 2020).

It is relatively straightforward to adapt these notions of elimination orderings to describe not only classical treewidth and generalized hypertree width, but also their threshold variants. In particular, by simply retracing the steps of the original proofs (Kloks, 1994; Fichte et al., 2018), one can show the following. Recall that for a CSP instance $\mathcal{I}$ and an integer d, we have defined $G^d_\mathcal{I}$ as the loaded graph obtained from the primal graph $G_\mathcal{I}$ of $\mathcal{I}$ by marking each vertex $v \in V(G_\mathcal{I})$ as light if $|D(v)| \leq d$ and heavy otherwise. Also, $H^d_\mathcal{I}$ is the loaded hypergraph obtained from the hypergraph $H_\mathcal{I}$ of $\mathcal{I}$ wherein we mark each hyperedge $F \in E(H_\mathcal{I})$ as light if $R(\gamma) \leq d$, where γ is the constraint corresponding to F, and we mark F as heavy otherwise.

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

Proof. We prove both parts of the statement simultaneously; we mainly describe the proof of part (1) and while doing so explain the differences to obtain part (2). First, we show the direction from a tree decomposition (resp. hypertree decomposition) to an elimination ordering. Let $\mathcal{I}$ be a CSP instance with threshold-d load-c treewidth k (resp. with generalized threshold-d load-c hypertree width k). Let (T, χ) be a load-c tree decomposition of width k for $G^d_\mathcal{I}$. For the case of hypertree width, let (T, χ, λ) be a generalized load-c hypertree decomposition. Let $n := |V(G^d_\mathcal{I})|$. Proceed as follows: Put $G'_0 = G^d_\mathcal{I}$ and $\mathcal{T}_0 = (T_0, \chi_0) := (T, \chi)$; respectively, put $\mathcal{T}_0 = (T_0, \chi_0, \lambda_0) := (T, \chi, \lambda)$. Then, for each $i = 1, 2, \ldots, n$ construct a graph G'_i, a vertex v_i, and a tree decomposition $\mathcal{T}_i = (T_i, \chi_i)$ (resp. a generalized hypertree decomposition $\mathcal{T}_i = (T_i, \chi_i, \lambda_i)$) as follows. Herein, we maintain the invariant that $\mathcal{T}_i$ is a load-c tree decomposition of width k for G'_i (resp. a load-c hypertree decomposition of width k for H_i, the hypergraph obtained from H by removing $v_1, \ldots, v_{i-1}$).

1. Pick an arbitrary leaf t in T_{i-1}. If each vertex in $\chi_{i-1}(t)$ occurs in the parent of t in T_{i-1}, remove t from T_{i-1}. Note that this results in another (generalized hyper-) tree decomposition of at most the same width and load. If t was removed, repeat this step.

2. After Step 1, in the picked leaf $t \in \mathcal{T}$ there is a vertex $v \in \chi_{i-1}(t)$ that occurs in no other bag of $\mathcal{T}_{i-1}$. Put $v_i := v$.

3. To obtain G'_i, take G'_{i-1}, remove v_i, and make $N_{G'_{i-1}}(v_i)$ a clique. To obtain $\mathcal{T}_i$, take $\mathcal{T}_{i-1}$ and remove v_i from all bags. Observe that this maintains our invariant because $N_{G_{i-1}}(v_i)$ is contained in the bag $\chi_{i-1}(t)$.

We claim that the elimination ordering $\prec$ on $V(G^d_{\mathcal{I}})$ induced by $v_1, v_2, \ldots, v_n$ has (cover) width k and for each v_i we have that $N_{G^d_{\mathcal{I},i-1}}(v_i) \cup \{v_i\}$ contains at most c heavy vertices (resp. for each v_i we have that $N_{G^d_{\mathcal{I},i-1}}(v_i) \cup \{v_i\}$ admits a hyperedge cover in $H_{\mathcal{I}}$ of size at most k and with at most c heavy hyperedges). Indeed, G'_i is equal to the graph G_i defined by $\prec$. In the case of tree decompositions, since $N_{G'_{i-1}}(v_i) \cup \{v_i\}$ is contained in the bag $\chi_{i-1}(t)$ in Step 3 and since $\mathcal{T}_{i-1}$ is a width-k load-c tree decomposition for G'_{i-1}, the ordering $\prec$ has width k and there are at most c heavy vertices in $N_{G_{i-1}}(v_i) \cup \{v_i\}$. Similarly, in the case of hypertree decompositions, since $N_{G'_{i-1}}(v_i) \cup \{v_i\}$ is contained in the bag $\chi_{i-1}(t)$ in Step 3 and since $\mathcal{T}_{i-1}$ is a width k load-c generalized hypertree decomposition for G'_{i-1}, the required cover of $N_{G'_{i-1}}(v_i)$ is given by $\lambda_{i-1}(t)$. Thus, the ordering $\prec$ has cover width k and $N_{G'_{i-1}}(v_i) \cup \{v_i\}$ admits a cover of size at most k with at most c heavy hyperedges. This completes the argument for the direction from tree decompositions to elimination orderings.

Now let $\prec$ be an elimination ordering for $G^d_{\mathcal{I}}$ with the properties promised in part (1) of the theorem (resp. in part (2)). Let $v_1, v_2, \ldots, v_n$ be the ordering of vertices of $G_{\mathcal{I}}$ induced by $\prec$ and let $G_1, G_2, \ldots, G_n$ be the corresponding graphs. Let G_{n+1} be the empty graph and let $\mathcal{T}_{n+1} = (T_{n+1}, \chi_{n+1})$ be a trivial tree decomposition for G_{n+1} wherein T_{n+1} consists of a single vertex t and the corresponding bag is empty. For hypertree decompositions we let $\mathcal{T}_{n+1} = (T_{n+1}, \chi_{n+1}, \lambda_{n+1})$ be an analogous hypertree decomposition, where additionally $\lambda_{n+1}(t) = \emptyset$. For each $i = n, n-1, \ldots, 1$ we construct a tree decomposition $\mathcal{T}_i = (T_i, \chi_i)$ for G_i (resp. a generalized hypertree decomposition for H_i, the hypergraph obtained from $H_{\mathcal{I}}$ by removing the vertices $v_1, v_2, \ldots, v_{i-1}$). Herein, we maintain the invariant that $\mathcal{T}_i$ is a (generalized hyper-) tree decomposition for G_i (resp. H_i) of width at most k and load at most c. At Step i, proceed as follows. Take $\mathcal{T}_{i+1}$ and find a node t in T_{i+1} such that the bag $\chi_{i+1}(t)$ contains $N_{G'_i}(v_i)$. Such a node exists, because $N_{G'_i}(v_i)$ is a clique in G_{i+1}. To obtain $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, add a new vertex t' as a child of t to T_{i+1} and define $\chi_i(t') := N_{G'_i}(v_i) \cup \{v_i\}$. Since $\prec$ has width k and there are at most c heavy vertices in $N_{G'_i}(v_i) \cup \{v_i\}$, we have that $\mathcal{T}_i$ is a load-c tree decomposition of width at most k for G_i. For hypertree decompositions, define also $\lambda_i(t')$ as the hyperedge cover in $H_{\mathcal{I}}$ of $N_{G'_i}(v_i) \cup \{v_i\}$ that has size at most k and contains at most c heavy hyperedges. Since H_i is a subhypergraph of $H_{\mathcal{I}}$, this cover is also a cover in H_i. Thus, $\mathcal{T}_i$ is a load-c hypertree decomposition of width at most k for H_i. This finishes the proof. □

A (significantly more complicated) elimination ordering characterizations of hypertree width has been obtained by Schidler and Szeider (2020, 2021). These, too, can be translated into characterizations 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.

Table 1: Overview of the algorithms that we use. The acronym ghtw stands for generalized hypertree width. We use *small* to indicate that the corresponding quantity is optimized using heuristic methods. We use *second* to indicate that the corresponding quantity was optimized as a secondary objective.

Parameter	Type	Name	Description
treewidth	Exact	`TW-X-Obl`	Minimum width, disregarding load.
		`TW-X-W→L`	Minimum width, load second.
		`TW-X-L→W`	Minimum load, treewidth second.
	Heuristic	`TW-H-Obl`	Small width, disregarding load.
		`TW-H-W→L`	Small width, load second.
		`TW-H-L→W`	Small load, treewidth second.
ghtw	Exact	`HT-X-Obl`	Minimum width, disregarding load.
		`HT-X-W→L`	Minimum width, load second.
		`HT-X-L→W`	Minimum load, width second.
	Branch & Bound	`HT-H-Obl`	Minimum (cover) width for heuristic tree decomposition, disregarding load.
		`HT-H-W→L`	Minimum (cover) width for heuristic tree decomposition, load second.
		`HT-H-L→W`	Minimum load for heuristic tree decomposition, (cover) width second.
	Greedy	`HT-G-Obl`	Small width, disregarding load.
		`HT-G-W→L`	Small width, load second.

7. Implemented Algorithms

We use classical exact and heuristic algorithms to compute tree decompositions and generalized hypertree decompositions and adapt them to take the load into account as described below. We call the algorithms without adaptions (load-) *oblivious*. These algorithms will bear the suffix *Obl* in the identifiers for the implemented algorithms that we introduce below. The adapted algorithms either minimize the width of the decomposition first (with heuristic or exact methods) and the load second, represented by suffix $W{\rightarrow}L$, or load first and width second, represented by suffix $L{\rightarrow}W$. Algorithms for treewidth are prefixed with TW and algorithms for (generalized) hypertree width are prefixed with HT. An overview over all algorithms can be found in Table 1.

All our algorithms are based on elimination orderings. A minimum-width elimination ordering without taking heavy vertices into account for a given graph can be computed using a SAT encoding (Samer & Veith, 2009); below we call this algorithm `TW-X-Obl`. This encoding can be extended to compute optimal generalized hypertree decompositions, by computing the covers for a tree decomposition of the primal graph (Fichte et al., 2018) using an SMT encoding, below denoted by `HT-X-Obl`. 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 the SMT approach to either compute a decomposition that minimizes the width first and the load second, that is,

a decomposition that has minimum width and, among all decompositions with minimum width, minimum load (leading to algorithms TW-X-W→L and HT-X-W→L). Or we use the SMT approach to compute a decomposition that minimizes the load first and the width second, that is, a decomposition that has minimum load and, among all decompositions with minimum load, 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 heuristic* constructs an ordering in a greedy fashion by choosing the i-th vertex, v_i, in the ordering among the vertices of minimum degree in the graph G_{i-1} as defined above, and yields decompositions with good width values overall (Bodlaender & Koster, 2011). Below we call this algorithm TW-H-Obl. 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; that is, it selects the i-th vertex, v_i, in the ordering as an arbitrary heavy vertex in G_{i-1} of minimum degree or, if G_{i-1} does not contain any heavy vertices, then it selects v_i to be an arbitrary vertex in G_{i-1} of minimum degree. This leads to decompositions with low load but possibly larger width. The latter heuristic (TW-H-W→L) maintains a bound $\ell \in \mathbb{N}$ on the target load of the decomposition, and selects the i-th vertex v_i in the ordering as an arbitrary vertex of minimum degree among all vertices in G_{i-1} that have 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 follow the general framework introduced by Dermaku, Ganzow, Gottlob, McMahan, Musliu, and Samer (2008). In particular, they begin by computing an elimination ordering for the primal graph using the min-degree heuristic, and then compute an edge cover for each bag. We use the same approach and employ two different methods to compute the covers: greedy and branch & bound (b&b).

The branch & bound heuristic computes an optimal edge cover for each bag. Although this approach optimally solves an in general NP-hard problem, it is viable in our data since the resulting Set Cover instances are comparatively easy. For convenience, let us call the size of the edge cover also its *width* and let the *load* of an edge cover be the number of heavy hyperedges contained in the cover. Note that minimizing the width (resp. load) of the cover corresponds to minimizing the width (resp. load) of the resulting decomposition. We use three different objectives: minimize the width of the cover only (HT-H-Obl), minimize width first and load second (HT-H-W→L), and minimize load first and width second (HT-H-L→W).

The greedy heuristic is a faster alternative to the branch & bound approach. The oblivious algorithm (HT-G-Obl) always adds the hyperedge that covers the most uncovered vertices of the current bag. Recall that this results in covers of width at most $(1 + \log n)$ times the minimum width of a cover, where n is the number of vertices (see, e.g., Chvatal (1979) or Theorem 1.11 by Williamson and Shmoys (2011)). We take the load into account by using the number of heavy hyperedges as a tie breaker when choosing the hyperedges (HT-G-W→L). This corresponds to a width first and load second strategy.

8. 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)

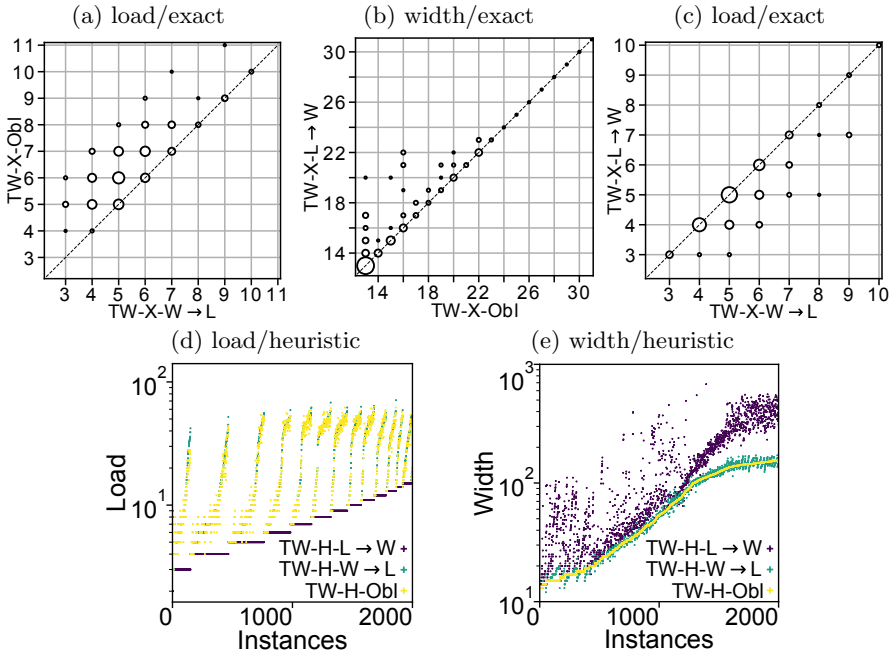

Figure 2: Exact and heuristic computations of tree decompositions: differences in values depending on the optimization strategy.

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 for the SAT and SMT instances we used *minisat 2.2.0* (Eén & Sörensson, 2003)[7] and *optimathsat 1.6.2* (Sebastiani & Trentin, 2020)[8]. The control code and heuristics use *Python 3.8.0*. Our code is freely available.[9] The nodes run *Ubuntu 18.04*. We used a 8 GB memory limit and a 2 hour time limit per instance.

Instances For threshold-d load-c tree decompositions we used 2788 instances from the *twlib*[10] benchmark set. For generalized threshold-d load-c hypertree decompositions we used the 3071 hyperbench (Fischl, Gottlob, Longo, & Pichler, 2019)[11] 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)).

7. http://minisat.se/
8. http://optimathsat.disi.unitn.it/
9. See https://github.com/ASchidler/htdsmt/tree/weighted and https://github.com/ASchidler/tw-sv.
10. http://www.cs.uu.nl/research/projects/treewidthlib/
11. http://hyperbench.dbai.tuwien.ac.at/

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; the number of instances with solutions is stated below.

Plots We use a specific type of scatter plot: 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 2a 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 7.

Treewidth Figures 2a to 2c 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 width. Further improvements in load can be obtained by using `TW-X-L→W`, as seen in Figure 2c. In Figure 2b we see that the trade-off (in terms of the width) required to achieve the optimal loads is often very small.

The results are different for heuristic methods. Figures 2d and 2e 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 2d 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 2e shows that `TW-H-L→W` tends to result in decompositions with much larger width 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 3a to 3c 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 3a), and `HT-X-L→W` can decrease the load even further (Figure 3a) while only slightly increasing the generalized hypertree width (Figure 3c).

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 3d. 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 vertex ordering). We can lower the load further by optimizing for load first as Figure 3f shows. Figure 3e shows that the resulting increase in width is about the same as the gain in load.

The results for the greedy heuristic look similar to the branch & bound results. Notably, the width is the same for most instances as shown in Figures 3e and h. The main difference is the slightly increased load as is shown in Figures 3d and 3g. This suggests that the greedy heuristic is a viable choice whenever a slightly higher load is acceptable.

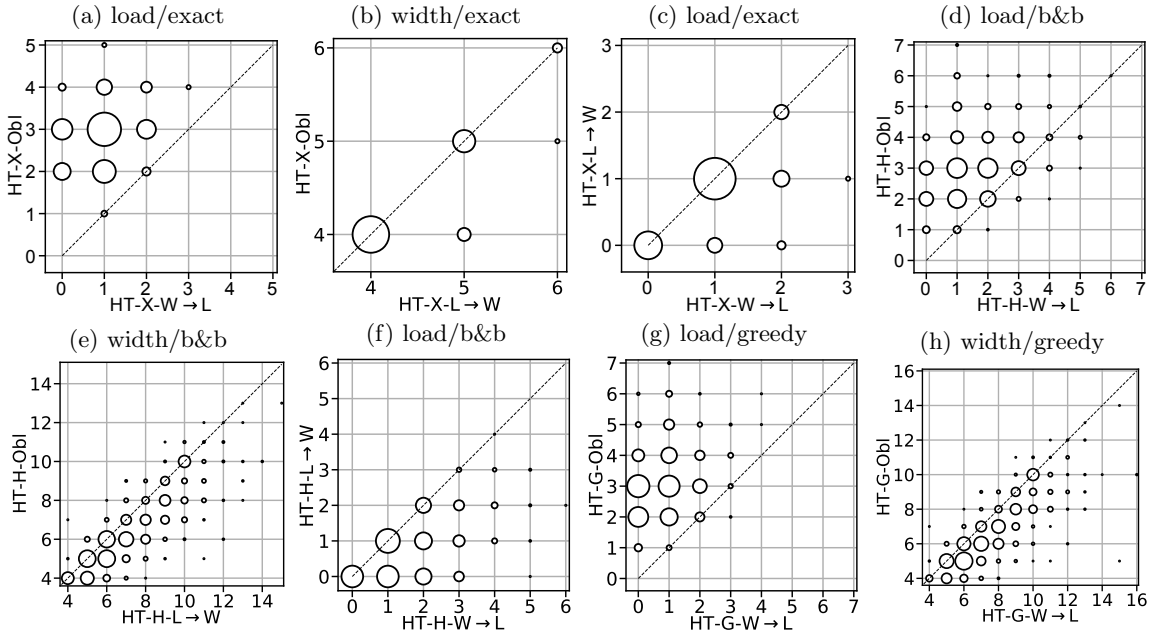

Figure 3: Exact and heuristic computations of generalized hypertree decompositions: differences in values depending on the optimization strategy.

9. 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 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.

A natural direction for future research is to explore how the concept of threshold treewidth can be adapted to CSPs in which variables may have infinite domains. On the one hand, several classes of such CSPs have been shown to be XP-tractable (Huang, Li, & Renz, 2013; Bodirsky & Dalmau, 2013) and even fixed-parameter tractable (Dabrowski, Jonsson, Ordyniak, & Osipov, 2021) with respect to the treewidth of the primal graph. This makes it interesting to attempt to further generalize these tractability results by using the threshold concept. On the other hand, in the finite-domain regime the potential "difficulty" induced by a domain can be captured straightforwardly by its size, however, it seems in the infinite-domain regime the difficulty of a domain has to be captured by different means. This is indicated when considering Mixed-Integer Linear Programs (MILPs) as CSPs: Checking the feasibility of MILPs is NP-hard but fixed-parameter tractable with respect to the number of integer variables (Lenstra, 1983). Thus the integer domains introduce the difficulty into

checking feasibility rather than the domain size alone. It thus seems important to capture the structure rather than the size of the domains. This would need a new approach.

Acknowledgments

Preliminary and shortened versions of the results presented in this submission appeared in the proceedings of IJCAI 2020 (Ganian, Schidler, Sorge, & Szeider, 2020). This article expands the exposition of that version by providing full proofs, detailed explanations especially including a more in-depth discussion of the applications of threshold treewidth, and an expanded experimental section.

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 projects P31336 and Y1329. 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 and by the Alexander von Humboldt Foundation. Main work of Manuel Sorge done while with University of Warsaw.

References

Abseher, M., Musliu, N., & Woltran, S. (2017). Improving the efficiency of dynamic programming on tree decompositions via machine learning. *Journal of Artificial Intelligence Research*, *58*, 829–858.

Bachoore, E., & Bodlaender, H. L. (2007). Weighted treewidth — algorithmic techniques and results. In *Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC '07)*, Vol. 4835 of *Lecture Notes in Computer Science*, pp. 893–903. Springer.

Bertele, U., & Brioschi, F. (1972). *Nonserial Dynamic Programming*. Academic Press, Inc., USA.

Bodirsky, M., & Dalmau, V. (2013). Datalog and constraint satisfaction with infinite templates. *Journal of Computer and System Sciences*, *79*(1), 79–100.

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

Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D., & Pilipczuk, M. (2016). A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, *45*(2), 317–378.

Bodlaender, H. L., & Koster, A. M. C. A. (2010). Treewidth computations I. Upper bounds. *Information and Computation*, *208*(3), 259–275.

Bodlaender, H. L., & Koster, A. M. C. A. (2011). Treewidth computations II. Lower bounds. *Information and Computation*, *209*(7), 1103–1119.

Brault-Baron, J., Capelli, F., & Mengel, S. (2015). Understanding model counting for beta-acyclic CNF-formulas. In *Proceedings of the 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, Vol. 30 of *LIPIcs*, pp. 143–156. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Carbonnel, C., & Cooper, M. C. (2016). Tractability in constraint satisfaction problems: a survey. *Constraints. An International Journal*, *21*(2), 115–144.

Carbonnel, C., Romero, M., & Zivný, S. (2018). The complexity of general-valued CSPs seen from the other side. In Thorup, M. (Ed.), *Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2018)*, pp. 236–246. IEEE Computer Society.

Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, *4*(3), 233–235.

Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., & Saurabh, S. (2015). *Parameterized Algorithms*. Springer.

Dabrowski, K. K., Jonsson, P., Ordyniak, S., & Osipov, G. (2021). Solving infinite-domain csps using the patchwork property. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 3715–3723. AAAI Press.

Dechter, R. (1999). Bucket elimination: a unifying framework for reasoning. *Artificial Intelligence*, *113*(1-2), 41–85.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N., & Samer, M. (2008). Heuristic methods for hypertree decomposition. In Gelbukh, A. F., & Morales, E. F. (Eds.), *Proceedings of the 7th Mexican International Conference on Advances in Artificial Intelligence(MICAI 2008)*, Vol. 5317 of *Lecture Notes in Computer Science*, pp. 1–11. Springer Verlag.

Diestel, R. (2012). *Graph Theory, 4th Edition*, Vol. 173 of *Graduate texts in mathematics*. Springer.

Downey, R. G., & Fellows, M. R. (2013). *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer Verlag.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Selected Revised Papers*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer Verlag.

Eiben, E., Ganian, R., Knop, D., & Ordyniak, S. (2019). Solving Integer Quadratic Programming via explicit and structural restrictions. In *Proceedings of the the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, pp. 1477–1484.

Fichte, J. K., Hecher, M., Lodha, N., & Szeider, S. (2018). An SMT approach to fractional hypertree width. In Hooker, J. N. (Ed.), *Proceedings of the 24rd International Conference on Principles and Practice of Constraint Programming (CP 2018)*, Vol. 11008 of *Lecture Notes in Computer Science*, pp. 109–127. Springer Verlag.

Fischl, W., Gottlob, G., Longo, D. M., & Pichler, R. (2019). Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019)*, pp. 464–480. ACM.

Fischl, W., Gottlob, G., & Pichler, R. (2018). General and fractional hypertree decompositions: Hard and easy cases. In den Bussche, J. V., & Arenas, M. (Eds.), *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2018)*, pp. 17–32. ACM.

Flum, J., & Grohe, M. (2006). *Parameterized Complexity Theory*, Vol. XIV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer Verlag, Berlin.

Freuder, E. C. (1982). A sufficient condition for backtrack-bounded search. *Journal of the ACM*, *29*(1), 24–32.

Ganian, R., Kim, E. J., Slivovsky, F., & Szeider, S. (2018). Sum-of-products with default values: Algorithms and complexity results. In Tsoukalas, L. H., Grégoire, É., & Alamaniotis, M. (Eds.), *Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, pp. 733–737. IEEE.

Ganian, R., Kim, E. J., Slivovsky, F., & Szeider, S. (2022). Sum-of-products with default values: Algorithms and complexity results. *Journal of Artificial Intelligence Research*, *73*, 535–552.

Ganian, R., Schidler, A., Sorge, M., & Szeider, S. (2020). Threshold treewidth and hypertree width. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI 2020)*, pp. 1898–1904. International Joint Conferences on Artificial Intelligence Organization.

Gottlob, G., Greco, G., & Scarcello, F. (2014). Treewidth and hypertree width. In Bordeaux, L., Hamadi, Y., & Kohli, P. (Eds.), *Tractability: Practical Approaches to Hard Problems*, pp. 3–38. Cambridge University Press.

Gottlob, G., Grohe, M., Musliu, N., Samer, M., & Scarcello, F. (2005). Hypertree Decompositions: Structure, Algorithms, and Applications. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2005)*, Vol. 3787 of *Lecture Notes in Computer Science*, pp. 1–15. Springer.

Gottlob, G., Leone, N., & Scarcello, F. (1999). On tractable queries and constraints. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA '99)*, Vol. 1677 of *Lecture Notes in Computer Science*, pp. 1–15. Springer.

Gottlob, G., Leone, N., & Scarcello, F. (2002). Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, *64*(3), 579–627.

Gottlob, G., Pichler, R., & Wei, F. (2010). Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, *174*(1), 105–132.

Gottlob, G., Scarcello, F., & Sideri, M. (2002). Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, *138*(1-2), 55–86.

Huang, J., Li, J. J., & Renz, J. (2013). Decomposition and tractability in qualitative spatial and temporal reasoning. *Artificial Intelligence*, *195*, 140–164.

Jégou, P., & Terrioux, C. (2017). Combining restarts, nogoods and bag-connected decompositions for solving CSPs. *Constraints An Int. J.*, *22*(2), 191–229.

Kask, K., Gelfand, A., Otten, L., & Dechter, R. (2011). Pushing the power of stochastic greedy ordering schemes for inference in graphical models. In Burgard, W., & Roth, D. (Eds.), *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, p. 54–60. AAAI Press.

Kloks, T. (1994). *Treewidth, Computations and Approximations*, Vol. 842 of *Lecture Notes in Computer Science*. Springer.

Köppe, M. (2012). On the complexity of nonlinear mixed-integer optimization. In Lee, J., & Leyffer, S. (Eds.), *Mixed Integer Nonlinear Programming*, Vol. 154, pp. 533–557. Springer.

Lenstra, H. W. (1983). Integer programming with a fixed number of variables. *Mathematics of Operations Research*, *8*(4), 538–548.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press.

Samer, M., & Szeider, S. (2010). Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, *76*(2), 103–114.

Samer, M., & Veith, H. (2009). Encoding treewidth into SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pp. 45–50.

Scarcello, F., Greco, G., & Leone, N. (2007). Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, *73*(3), 475–506.

Schidler, A., & Szeider, S. (2020). Computing optimal hypertree decompositions. In Blelloch, G., & Finocchi, I. (Eds.), *Proceedings of the 22nd Workshop on Algorithm Engineering and Experiments (ALENEX 2020)*, pp. 1–11. SIAM.

Schidler, A., & Szeider, S. (2021). Computing optimal hypertree decompositions with SAT. In Zhou, Z. (Ed.), *Proceeding of IJCAI-21, the 30th International Joint Conference on Artificial Intelligence*.

Schiex, T., Fargier, H., & Verfaillie, G. (1995). Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI '95)*, pp. 631–639.

Schrijver, A. (1999). *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley.

Sebastiani, R., & Trentin, P. (2020). OptiMathSAT: A tool for optimization modulo theories. *Journal of Automated Reasoning*, *64*(3), 423–460.

Williamson, D. P., & Shmoys, D. B. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.

Zivny, S. (2012). *The Complexity of Valued Constraint Satisfaction Problems.* Cognitive Technologies. Springer.