The Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-24)

SAT-Based Tree Decomposition with Iterative Cascading Policy Selection

Hai Xia, Stefan Szeider

Algorithms and Complexity Group, TU Wien, Austria
{hxia,sz}@ac.tuwien.ac.at

Abstract

Solvers for propositional satisfiability (SAT) effectively tackle hard optimization problems. However, translating to SAT can cause a significant size increase, restricting its use to smaller instances. To mitigate this, frameworks using multiple local SAT calls for gradually improving a heuristic solution have been proposed. The performance of such algorithmic frameworks heavily relies on critical parameters, including the size of selected local instances and the time allocated per SAT call.

This paper examines the automated configuration of the treewidth SAT-based local improvement method (TW-SLIM) framework, which uses multiple SAT calls for computing tree decompositions of small width, a fundamental problem in combinatorial optimization. We explore various TW-SLIM configuration methods, including offline learning and real-time adjustments, significantly outperforming default settings in multi-SAT scenarios with changing problems.

Building upon insights gained from offline training and real-time configurations for TW-SLIM, we propose the iterative cascading policy—a novel hybrid technique that uniquely combines both. The iterative cascading policy employs a pool of 30 configurations obtained through clustering-based offline methods, deploying them in dynamic cascades across multiple rounds. In each round, the 30 configurations are tested according to the cascading ordering, and the best tree decomposition is retained for further improvement, with the option to adjust the following ordering of cascades. This iterative approach significantly enhances the performance of TW-SLIM beyond baseline results, even within varying global timeouts. This highlights the effectiveness of the proposed iterative cascading policy in enhancing the efficiency and efficacy of complex algorithmic frameworks like TW-SLIM.

Introduction

Over the last two decades, SAT-solver technology has made tremendous progress; SAT instances with up to over a million variables and clauses can be solved routinely (Fichte et al. 2023). However, for many combinatorial optimization problems, the encoding to SAT entails a significant blow-up in size (cubic or worse), significantly limiting the feasible instance size of the combinatorial problem. To make SAT still applicable to large combinatorial problem instances, researchers have developed new algorithmic frameworks where SAT solvers are called multiple times, where each call deals only with a small part of the combinatorial instance. *SAT-based Local Improvement (SLIM)*, a particular structure-driven form of *Large Neighborhood Search (LNS)* (Pisinger and Ropke 2010), is such an algorithmic framework where multiple local SAT calls improve a global heuristic solution (Lodha, Ordyniak, and Szeider 2016; Fichte, Lodha, and Szeider 2017; Lodha, Ordyniak, and Szeider 2017; Peruvemba Ramaswamy and Szeider 2021a,b; Schidler and Szeider 2021; Ramaswamy and Szeider 2022; Schidler 2022; Kulikov, Pechenev, and Slezkin 2022; Reichl, Slivovsky, and Szeider 2023).

However, the performance of algorithmic frameworks like SLIM that rely on multiple SAT calls heavily depends on several critical parameters: how large the selected local part should be and how much time each individual SAT call should have at its disposal. Moreover, since the instance evolves and changes during the solving time, dynamic aspects must be considered, such as when to switch from one configuration to another and in what order.

Although there is a large bulk of work on automated algorithm configuration and selection (see, e.g., Schede et al.'s (2022) survey), there needs to be a rigorous study on techniques to configure such a complex algorithmic framework like SLIM that involves multiple SAT calls and requires the consideration of dynamic aspects. In this paper, we provide such a study, showcasing the widely studied problem of *finding a small-width tree decomposition of a graph* (MINTW). We used the TW-SLIM approach for this problem by Fichte, Lodha, and Szeider (2017), who followed the SLIM paradigm to compute small-width tree decompositions for large graphs based on repeated SAT calls.

We developed and compared a wide range of approaches to configure TW-SLIM.

As a baseline, we develop approaches based purely on offline learning on training data and purely on a real-time configuration, respectively. These relatively standard approaches significantly improve performance over the hand-tuned default configuration of TW-SLIM.

Based on these encouraging preliminary results, we propose the *iterative cascading policy* (*ICP*), a hybrid approach that combines offline and real-time methods in a new way. The iterative cascading policy uses a pool of

30 configurations obtained by a constrained clustering-based offline approach and deploys them along a dynamic cascade. This goes beyond static cascading portfolios (Eiben et al. 2019; Roussel 2012; Streeter 2018) that run individual configurations along one static cascade. Iterative cascading is arranged in several rounds, where a cascade of configurations is tried in each round. The best-so-far tree decomposition is only replaced at the end of the round when the best tree decomposition of the current round is found. Each round can change the linear ordering for its cascade by considering the updated features of instances. According to the performance of the configurations among different clusters of instances, the algorithm switched to the most suitable cascading ordering and updated the tree decomposition for further improvements.

We can boost the performance of TW-SLIM significantly beyond the baseline results with iterative cascading. We conducted our experiments over a comprehensive set of over 3000 benchmark graphs from various real-world applications. We randomly split these graphs $80 : 20$ into a training set on which we performed the offline training and a test set on which we report the observed performance. The primary objective function counts the total sum of improvements ($\mathsf{T}\Sigma$) of treewidth, a good proxy for the number of instances whose treewidth could be reduced ($\mathsf{T}\#$).

The standard hand-tuned configuration used by Fichte, Lodha, and Szeider (2017) gives $\mathsf{T}\Sigma = 398$ in the default 7800-second timeout they used for their experiments. With the clustering-based offline configuration, this value increases to 456 and 500, depending on whether the configuration is selected with AutoFolio (Lindauer et al. 2015) or the constrained-clustering model integrated by us. The latter, when applied with our devised dynamical adaptation (i.e., a new configuration is chosen according to the dynamic changes of the instance), gets up to $\mathsf{T}\Sigma = 619$, already a significant gain over the original hand-tuned configuration. Finally, our new iterative cascading policy can boost the value to 728.

We provide empirical data with shorter global timeouts, from 100 to 7800 seconds. For shorter timeouts, the ordering with a cascade becomes even more critical for good performance, and it makes sense to reduce the number of configurations within a cascade to cover the space of configurations faster. With these experiments, we can differentiate between variants of the iterative cascading policy and see that those that include offline training and work with smaller cascades have an advantage if less time is available for real-time improvement. Still, the iterative cascading policy has a clear lead over the other more conventional approaches, even for the shorter timeouts.

Related Work

Many algorithm configuration tools have been proposed to tune hyperparameters. The overview table in Schede et al.'s comprehensive survey (2022) shows that only 2 of the 42 considered configuration tools adjust the configuration dynamically during the algorithm's run, and the others are static. From the aspect of the training setting, there are also only 2 with real-time training. In general, most algorithms regard the configuration problem as a black-box optimization problem and use the offline paradigm (Ansótegui, Sellmann, and Tierney 2009; Hutter, Hoos, and Leyton-Brown 2011; Lindauer et al. 2021; López-Ibáñez et al. 2016), where the configurator receives all training instances as the tuning begins and then searches for a suitable configuration. Only few dynamic methods were proposed to adjust the real-time configurations for better performance (Fitzgerald, Malitsky, and O'Sullivan 2015; Fitzgerald et al. 2014). These configurators received a stream of changing problems and were required to solve dynamic problem instances with suitable configurations. To configure algorithms for instances with different features, some researchers also proposed instance-specific algorithm selection methods (Kadioglu et al. 2010; Lindauer et al. 2015; Xu et al. 2008), where a (portfolio) classifier learned to predict the best algorithms (configurations) by features.

Cascading portfolio scheduling arranges diverse policies or configurations in a linear order which optimizes a sequential run on training instances (Eiben et al. 2019; Roussel 2012; Streeter 2018). Our setting is special, given the nature of our instances, consisting of the input graph and a heuristically computed initial tree decomposition. The configuration needs to consider features of both parts, where one part (the graph) remains steady through the solving time, and the other part (the tree decomposition) is subject to change. Most of the studies in the literature focus on static problems with significantly shorter timeouts, whereas we work with a global timeout (7800 seconds) which is typical for SLIM algorithms (Kulikov, Pechenev, and Slezkin 2022; Lodha, Ordyniak, and Szeider 2016, 2017; Peruvemba Ramaswamy and Szeider 2021a,b; Ramaswamy and Szeider 2022, 2020; Reichl, Slivovsky, and Szeider 2023; Schidler 2022; Schidler and Szeider 2021).

SLIM for Tree Decompositions

In this section, we introduce some basic relevant concepts on graphs and tree decompositions and outline the SLIM approach to treewidth computations.

All graphs considered are finite and simple. We define a graph G in terms of its set $V(G)$ of vertices and set $E(G)$ of edges. We denote an edge between vertices $u, v \in V(G)$ by uv or, equivalently, vu. A *subgraph* H of G *induced* by a set $X \subseteq V(G)$ has $V(H) = X$ and $E(H) = \{\, uv \in E(G) \mid u, v \in X \,\}$.

A *tree decomposition* (TD) of a graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a tree and χ is a mapping that assigns each tree node $t \in V(T)$ a subset $\chi(t) \subseteq V(G)$ such that (i) for all $uv \in E(G)$ there is some $t \in V(T)$ with $u, v \in \chi(t)$, and (ii) for each $v \in V(G)$ the set $\{\, t \in V(T) \mid v \in \chi(t) \,\}$ indices a connected subtree of T. The sets $\chi(t)$, $t \in V(T)$ are called *bags*. The *width* $w(\mathcal{T})$ of $\mathcal{T}$ is $\max_{t \in V(T)} |\chi(t)| - 1$, and the *treewidth* of G is the smallest width over all its tree decompositions (Bodlaender 1993; Kloks 1996). We consider the optimization problem MINTW, which takes as input a graph G and asks for a tree decomposition of G of the smallest width. MINTW is NP-hard (Arnborg, Corneil, and Proskurowski 1987). Because of its intractability, exact algorithms apply only to small graphs (Bannach, Berndt, and

Ehlers 2017; Samer and Veith 2009; Tamaki 2022), whereas for large graphs, heuristics are used (Abseher, Musliu, and Woltran 2017; Bodlaender and Koster 2010) that compute a possibly suboptimal upper bound for the treewidth.

Fichte, Lodha, and Szeider (2017) proposed an algorithm for MINTW using the SAT-based local improvement (SLIM) metaheuristic. We refer to this algorithm as TW-SLIM. Subsequently, we outline its workflow, thereby introducing relevant parameters.

Let G be the input graph to MINTW. First, an *initial tree decomposition* $\mathcal{T} = (T, \chi)$ is computed with a heuristic method. Next, the following improvement step is repeatedly performed: A subtree S of T is selected such that the size of $X = \bigcup_{t \in V(S)} \chi(t)$ does not exceed a *local budget* (parameter: lb). Let $\mathcal{S} = (S, \chi_S)$ be the *local tree decomposition* with $\chi_\mathcal{S}$ being the restriction of χ to S. Now $\mathcal{S}$ is a tree decomposition of the subgraph $G_\mathcal{S}$ of G induced by X with $w(\mathcal{S}) \leq w(\mathcal{T})$. To ensure *replacement consistency* (that we can substitute $\mathcal{S}$ in $\mathcal{T}$ with a tree decomposition of smaller width), we add certain edges to $G_\mathcal{S}$: We obtain the *augmented local graph* $G^*_\mathcal{S}$ from $G_\mathcal{S}$ by adding for each $st \in E(T)$ such that $s \in V(S)$ and $t \notin V(S)$, all the edges uv with $u, v \in \chi(s) \cap \chi(t)$. As shown by Fichte, Lodha, and Szeider (2017), $\mathcal{S}$ is still a tree decomposition of $G^*_\mathcal{S}$, and more importantly, we can replace in $\mathcal{T}$ the local tree decomposition $\mathcal{S}$ with any new tree decomposition $\mathcal{S}^*$, resulting in a tree decomposition $\mathcal{T}^*$ (we explain below how $\mathcal{S}^*$ is obtained). For $w = \max_{t \in V(T) \setminus V(S)} |\chi(t)| - 1$, we have $w(\mathcal{T}^*) \leq \max(w, w(\mathcal{S}^*))$ (Fichte, Lodha, and Szeider 2017, Observation 3); thus, by reducing the with of local tree decompositions, we can eventually reduce the width of the global tree decomposition.

Since $G^*_\mathcal{S}$ is sufficiently small (its number of vertices is at most lb), we can compute its treewidth exactly using a SAT encoding. That is, we set $k = w(\mathcal{S})$ and generate a propositional formula $F(G^*_\mathcal{S}, k-1)$, which is satisfiable if and only if the treewidth of $G^*_\mathcal{S}$ is at most $k-1$, and feed this to a SAT solver with a specified *SAT timeout* (parameter st). The limit st is important, as we want to run the solver long enough to return a satisfying assignment, but stop the solver before it determines unsatisfiability, which takes an order of magnitude longer. This way, we can save precious time that is better used by trying different local instances.

If the SAT solver determines that $F(G^*_\mathcal{S}, k-1)$ is satisfiable, we try a further reduction $F(G^*_\mathcal{S}, k-2)$, and so forth, until we reach a *local timeout* (parameter: lt). From the last satisfiable SAT call, we can read off the new tree decomposition $\mathcal{S}^*$ that we insert in $\mathcal{T}$ instead of $\mathcal{S}$, giving us the tree decomposition $\mathcal{T}^*$ of G.

If the SAT solver does not determine that $F(G^*_\mathcal{S}, k-1)$ is satisfiable (either by determining its unsatisfiability or reaching the st limit), we can run the solver on $F(G^*_\mathcal{S}, k)$, which is guaranteed to be satisfiable since the treewidth of $G^*_\mathcal{S}$ is at most k. However, the satisfying assignment found will most likely give rise to a tree decomposition $\mathcal{S}^*$ that is different from $\mathcal{S}$. Hence, replacing $\mathcal{S}$ with $\mathcal{S}^*$ in $\mathcal{T}$ will not reduce the treewidth of $\mathcal{T}$ but will *shuffle* it (whether a shuffle takes place is controlled by a flag parameter: sf), so that future attempts for improvement will have a better chance to escape a local optimum.

Parameter name	Parameter meaning
local budget (lb)	budget of vertices for SLIM
local time (lt)	timeout for the local improvement
SAT time (st)	call timeout for the SAT solver
switch flag (sf)	replace $G_\mathcal{S}$ if $w(G^*_\mathcal{S}) = w(G)$

Table 1: Key parameters and their meaning

Figure 1 illustrates one improvement step of TW-SLIM. TW-SLIM terminates if either a *global timeout* (parameter: gt) is reached or there have been a certain number of *non-improvement* steps (parameter: ni). Table 1 shows the key parameters that we configure for TW-SLIM.

Experimental Setup

In this section, we present the setting of experiments from different aspects. We provide an external link for source code and detailed results (Xia and Szeider 2023).

Instances

To obtain a comprehensive evaluation, we collected a large set of benchmark graphs from various collections (see Table 2). Since the proposed algorithms are aimed at TDs of large graphs that exact methods cannot solve, the graphs with fewer than 100 vertices are not considered. Graphs with over 10^6 vertices are also filtered out because even the heuristic methods cannot work with them due to memory overflows. This restriction results in a total of 3331 instances. After the whole data set is split randomly into a training set (80%) and a test set (20%), the training set and test set have 2664 and 667 instances, respectively.

Setup

All experiments are carried out on a Linux (Ubuntu 18.04.6 LTS) Sun Grid Engine cluster with 3 nodes, where each node has two AMD EPYC 7402 CPUs (each has 24 cores with a frequency of 2.80GHz). Due to the limitation in compatibility, we use different Python versions for different parts of the employed components. Components related to TW-SLIM and AutoFolio run on Python 2.7.5 and Python 3.5, respectively. All other components run on Python 3.9.12. For a fair comparison, we set the same global search timeout to 7800 seconds as recommended by the original paper

Instances	URLs (https://)
functions	sdcc.sourceforge.net
PACE16	pacechallenge.org/2016/treewidth
PACE17	pacechallenge.org/2017/treewidth
UAI	www.ics.uci.edu/~dechter/software.html
Roadnet	www.diag.uniroma1.it/challenge9
TWlib	webspace.science.uu.nl/~bodla101/treewidthlib

Table 2: Data sources of all considered instances

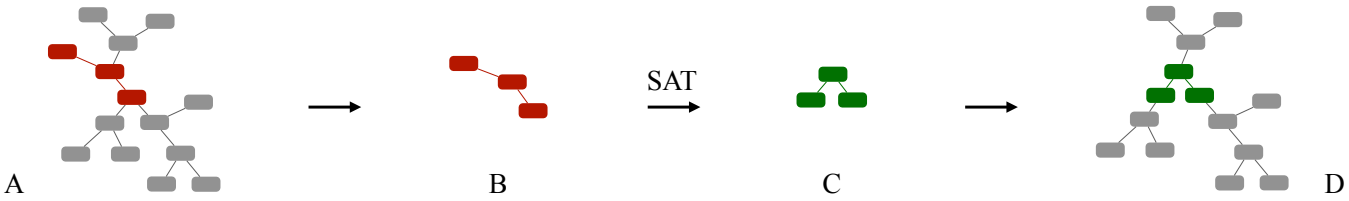

Figure 1: One single SAT-based improvement step within TW-SLIM. From A to B: a subtree that induces a local instance whose size does not exceed the local budget is selected; from B to C: an improved tree decomposition for the local instance is computed with a SAT call; from C to D, the original subtree is replaced by an improved subtree according to replacement consistency properties.

of TW-SLIM (Fichte, Lodha, and Szeider 2017), the heuristic for computing the initial TD to *HTD* (version: v0.9.5-beta) (Abseher, Musliu, and Woltran 2017), the local solver to *Jdrasil* (Bannach, Berndt, and Ehlers 2017), and the SAT solver to *Glucose* (Audemard and Simon 2018).

Optimization Objectives

For an instance $(G, \mathcal{T})$ consisting of the input graph G and the heuristically computed initial tree decomposition $\mathcal{T}$, let $\mathcal{T}^*$ be the tree decomposition at the end of a TD-SLIM run. If $w(\mathcal{T}^*) < w(\mathcal{T})$, then the run was successful, as the width of the initial tree decomposition was improved. One measure for the performance of a configuration is to count the *total number of improved instances* (T#). A more fine-grained measurement takes the *total sum of improvements* (TΣ). Clearly TΣ > T# as an improved instance contributes at least 1 to T#. It turned out that taking TΣ as the objective yields excellent results also for T#, even better than with T# as the objective (this is plausible since TΣ gives more detailed feedback to the configurator than TΣ). Therefore, we take TΣ as the objective throughout the experiments, but also report on T#.

We would like to note that we always first run the heuristic and measure the improvement SLIM gains over the initial heuristic solution. Thus, for an instance that we consider easy, the heuristic could provide a reasonable upper bound for the treewidth, which made the instance more challenging for TW-SLIM to improve. For many applications of tree decompositions, like exact probabilistic reasoning, the worst-case time complexity is exponential in the treewidth, which means that even tiny reductions in the treewidth yield significant performance improvements.

Offline Configuration

In this section, we introduce a series of offline configurators as baselines for further improvements.

Optimizing for One Configuration over the Entire Data Set (*SB-all*)

Given the set of training instances, we use the state-of-the-art algorithm configuration tool SMAC (Hutter, Hoos, and Leyton-Brown 2011) to search for the most promising TW-SLIM configuration. Then, the performance of the configuration is examined on test instances. The setting of SMAC throughout our experiments is listed in Table 3.

Parameter name	Value
configuration	lb $\in [50, 300]$, 100 as default lt $\in [90, 5000]$, 1800 as default st $\in [90, 5000]$, 900 as default sf $\in \{0, 1\}$, 1 as default
wallclock-limit	48 hours
cutoff-time	4 hours
memory limit	300 GB
objective	total improvements of widths
limit_resources	False
model type	Gaussian process
acquisition function	expected improvement
random state	42

Table 3: The setting of SMAC

Algorithms	Training set	Test set
hand-tuned TW-SLIM	1361/640	398/182
SB-all	**1674**/683	**457**/175

Table 4: Performance comparison between original hand-tuned TW-SLIM and *SB-all* on 2664 training instances and 667 test instances. All results have the format TΣ/T#.

Table 4 shows the performance of the systematically hand-tuned TW-SLIM (the default configuration of TW-SLIM (Fichte, Lodha, and Szeider 2017)) and the automatically-tuned TW-SLIM (*SB-all*). On the training instances, the automatically-tuned TW-SLIM exhibits significant improvements on both measurements: TΣ increased by 313 and T# increased by 43. On the test instances, we observe TΣ increased by 59 and T# decreased by 7.

Clustering-Based Offline Configuration (*CC*)

For the instance-specific configuration of TW-SLIM, we consider the nine features listed in Table 5, where seven features describe properties of the TD (and may change during the optimization process), and the other two features describe properties of the input graph (that remains constant during the optimization process). We utilize *constrained K-means* (Bradley, Bennett, and Demiriz 2000) to cluster training instances into clusters because it can control the number

Feature	Aspect	Range
Number of bags	TDs	$[2, 418434]$
Largest bag size	TDs	$[2, 3082]$
Number of vertices	Graphs	$[100, 468913]$
Sum of bag sizes	TDs	$[300, 2404836]$
Smallest bag size	TDs	$[2, 400]$
Sum of out degrees	TDs	$[1, 418433]$
Number of leaves	TDs	$[1, 159582]$
Depth	TDs	$[1, 306]$
Number of edges	Graphs	$[100, 863026]$

Table 5: Selected features of problem instances

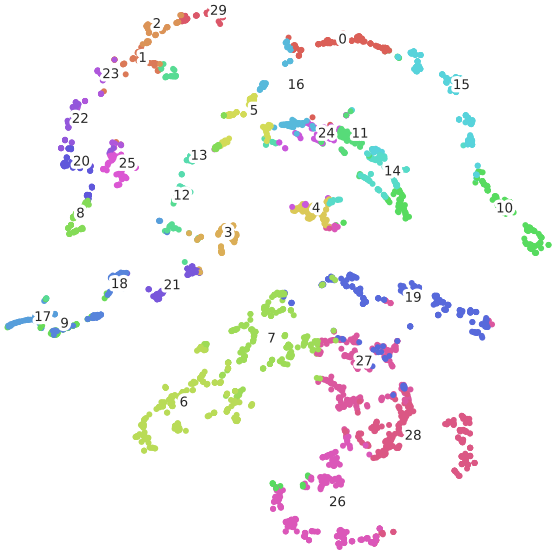

Figure 2: The distribution of training instances is visualized by T-NES. Different colors of dots are instances within different clusters; Numbers (0 to 29) are 30 clustering labels.

of instances within each cluster. We set the number of clusters to 30, and add the minimum (50) and maximum (200) number of instances in each cluster as clustering constraints. Therefore, we can balance the distribution of training instances, which is also the issue reported by previous algorithms (Kadioglu et al. 2010). Therefore, SMAC can search for promising configurations among different clusters of instances separately. We can use the same clustering model to predict the best configurations for test instances by classifying their features into the most similar clusters. We refer to this method by *CC-sta* (for clustering-clustering-static).

To examine the effectiveness of our constrained clustering with the selected nine features, we apply the high-dimensional visualization tool, t-distributed stochastic neighbor embedding (Hinton and Roweis 2002) (T-SNE), to visualize clusters of training instances learned by constrained K-means. From the distribution of different clusters in Figure 2, most instances with similar features are clustered into the same cluster, which implies the effectiveness of the constrained K-means clustering.

Algorithms	Training set	Test set
CA with 1-hour training	✗	456/185
CA with 2-hour training	✗	456/185
CA with 6-hour training	✗	408/169
SB-all	1674/683	457/175
CC-sta	**2143**/822	**500**/190
SB	1789/718	468/185
VB	**2621**/826	**738**/222

Table 6: Performance comparison between *SB-all*, *CC-sta*, and *CA* on 2664 training instances and 667 test instances. All results have the format TΣ/T#.

As a comparison, we utilize the state-of-the-art portfolio algorithm selector AutoFolio (Lindauer et al. 2015), which tries to predict the best configurations for new instances according to their features. We refer to this method as Cluster-AutoFolio (*CA*). We use the same training instances to obtain the feature table and the same configurations (one from each cluster) for the performance table. Different training times are examined, and the performance of *CA* with 1, 2, and 6 hours of training time are given in Table 6, where both the theoretical performances of the single best (SB) and the virtual best (VB) are calculated according to the survey (Schede et al. 2022).

From the results in Table 6, *CA* with a 1 or 2-hour training exhibits the best performance among different training times. Somewhat surprisingly, the performance of *CA* is worse than *CC-sta* and even worse than the performance of the theoretical SB. However, *CC-sta* outperforms *SB-all* significantly with respect to TΣ, bridging 43% and 11% of the gap between the SB and the VB on the training set and test set, respectively. With respect to T#, *CC-sta* can bridge the gap between the SB and the VB by 96% and 14% on the training set and test set, respectively.

Table 6 also shows the performance of *SB-all* and *CC-sta* on the training set and test set. The settings of SMAC for the whole set of training instances and a cluster of training instances are the same. For the former task, SMAC had access to 80 cores, and for the latter task, SMAC was given 8 cores. For both measurements TΣ and T#, *CC-sta* outperforms the *SB-all* SMAC. Compared with the performance of the hand-tuned TW-SLIM, *CC-sta* results in increases in TΣ (102) and T# (8), and there is also an increase compared with *SB-all*: TΣ (43) and T# (15).

Dynamic Configuration Selection

Offline methods must always obtain some training instances first and then apply the policy learned from the training set to a similar test set. However, in the SLIM context, optimized subgraphs result in changing instances. Hence, applying the same configuration within the whole optimization will be ineffective. Accordingly, we propose the dynamic variant *CC-dyn* of *CC-sta* to adapt to the dynamics of the problem during the run. In *CC-dyn*, if the configuration predicted by the clustering model can improve a TD, the next configuration will be selected by the same model again according

to the updated features. Otherwise, if a configuration fails to improve a TD further, a different configuration will be selected randomly from the pool consisting of the 30 configurations until the global timeout is reached. As a comparison, we also use the state-of-the-art offline configurator SMAC in an online fashion, where SMAC tunes during runtime for the specific instance within different wallclock limits (the timeout including configuring and tree decomposition). This new and atypical use of SMAC makes sense in a setting with a longer timeout to accommodate real-time tuning. Except for the wallclock limit, other corresponding parameters are also set according to Table 3. We refer to the per-instance SMAC algorithm as *SB-one*.

To examine the performance of *CC-dyn*, we set different global timeouts from 100 to 7800 seconds, which are also the wallclock limits used by *SB-one*. From Table 7, within the default timeout (7800 seconds) used by TW-SLIM, *CC-dyn* increases $\mathsf{T}\Sigma$ significantly by 119 from *CC-sta*. Compared with *SB-all* and hand-tuned TW-SLIM, $\mathsf{T}\Sigma$ is improved by 56% and 35%, respectively. As the timeout decreases below 500 seconds, *CC-dyn* deteriorates so heavily that its performance is worse than *CC-sta*. The inefficiency is probably caused by the "trial-and-error" application of some configurations during the running, which can give an impetus for long-term optimization. The contrast between *CC-dyn* and *SB-one* is similar, but *SB-one* can outperform more significantly within 500 seconds of timeout: with less than 100 seconds, *SB-one* obtains around 40% more $\mathsf{T}\Sigma$ than both *CC-sta* and *CC-dyn*, because we set the good-enough default configuration ($\mathsf{lb} = 50$, $\mathsf{lt} = 1800$, $\mathsf{st} = 900$, $\mathsf{sf} = 1$) for *SB-one* according to the original hand-tuned TW-SLIM, and *SB-one* can use the default configuration even at the beginning. However, *CC-sta* and *CC-dyn* have to run with a better configuration only after trying different configurations. Besides, improvements are calculated for configurations with a 7800-second timeout when promising configurations are searched among clusters. Therefore, within only 100 seconds, it is possible that the recommended configuration is not as good as it is expected.

In general, *CC-dyn* can improve the performance of *CC-sta* by applying the adaptive selection policy within the default timeout (7800 seconds). With shorter optimization timeouts, the performance of *CC-dyn* may deteriorate due to the exploration of promising configurations. Therefore, building upon the insights we gained so far through our systematic study, we will propose in the next section the iterative cascading policy for applying the configurations more efficiently to boost the performance of TW-SLIM in various timeout scales.

Iterative Cascading Policy

In this section, we introduce our iterative cascading policy, which combines offline and dynamic methods in a new way, boosting the overall performance given all different timeouts.

Algorithm Framework

The starting point for this policy is cascading portfolio scheduling (Eiben et al. 2019; Streeter 2018), where one linear ordering of m configurations (the cascade) is computed offline; the configurations are then run sequentially following this order until a timeout is reached. Differently, we can use the $m = 30$ configurations obtained with the clustering method (see Section 5) to form such a cascade, and we can adaptively assign different cascading orders to different instances based on their updated features, according to the average performance on the training data from different clusters. Meanwhile, we sort the configurations by calculating their improvements over the training instances, and we can use two metrics of improvements ($\mathsf{T}\Sigma$ and $\mathsf{T}\#$) for sorting.

Sometimes, the global timeout is reached before all the m configurations of the cascade have been run, so running the most promising configurations first is essential. If the first cascade could be completed, we take the tree decomposition of the lowest width obtained within all the m configurations and start a new round with this as the initial tree decomposition. The static version of *ICP* (*ICP-sta*) now uses the same cascade in the next round and repeats this process until the global timeout is reached.

The dynamic version of *ICP* (*ICP-dyn*) also proceeds in rounds. However, we formulate 30 different cascades according to the performance of configurations on the 30 clusters of training instances (the same clusters we used in *CC*). Then, every time a new round begins, the clustering model (the same as used in *CC*) is applied to select one of the 30 cascades by the updated instance features. This way, the cascade is always up-to-date and adjusted in

Algorithm 1: Dynamic *ICP*

Input: Initial graph G, heuristic tree decomposition $\mathcal{T}$, a set of cascading orderings of configurations $\mathbb{S} = \{\mathbb{C}_1, \mathbb{C}_2, ..., \mathbb{C}_n\}$ where $\mathbb{C}_i = [C_{i,1}, C_{i,2}, ..., C_{i,m}]$, algorithm selector Classifier(), feature extractor Extractor()
Output: Final tree decomposition $\mathcal{T}^*$

$i \leftarrow 0$, $\mathcal{T}_b \leftarrow \mathcal{T}$, $\mathcal{T}_c \leftarrow \mathcal{T}$ // initialization;
$\mathbb{F} \leftarrow$ Extractor$(G, \mathcal{T}_c)$ // get the updated features;
$l \leftarrow$ Classifier$(\mathbb{F})$ // get the clustering label;
while *Not timeout* **do**
  if $i = m$ **then**
    $i \leftarrow 0$;
    $\mathcal{T}_b \leftarrow \mathcal{T}_c$ // update the problem instance;
    $\mathbb{F} \leftarrow$ Extractor$(G, \mathcal{T}_c)$;
    $l \leftarrow$ Classifier$(\mathbb{F})$;
  end
  else
    $i \leftarrow i + 1$;
    $\Delta t \leftarrow$ TW-SLIM$_{C_{l,i}}(G, \mathcal{T}_b)$ // apply $C_{l,i}$;
    if $w(\mathcal{T}_t) < w(\mathcal{T}_c)$ **then**
      $\mathcal{T}_c \leftarrow \mathcal{T}_t$ // update the best TD of this round;
    end
  end
end
$\mathcal{T}^* \leftarrow \mathcal{T}_c$;

Algorithm	TS	CA	100 seconds	500 seconds	1000 seconds	3000 seconds	7800 seconds
Hand-tuned	O	S	✗	✗	✗	✗	398/182
CA	O	S	✗	✗	✗	✗	456/185
SB-all	O	S	✗	✗	✗	✗	457/175
CC-sta	O	S	234/132	395/165	418/172	445/179	500/190
CC-dyn	O	D	223/128	386/163	493/179	571/197	619/200
SB-one	R	S	319/160	457/177	490/184	534/190	587/194
ICP-sta (TΣ)	O	D	361/168	504/194	562/203	**704**/220	**728**/219
ICP-sta (T#)	O	D	334/161	503/190	541/196	672/212	691/213
ICP-dyn (TΣ)	O	D	296/155	471/182	541/194	658/209	712/215
ICP-dyn (T#)	O	D	338/163	**506**/188	**574**/200	682/212	710/213
ICP-sta (TΣ) (ub)	O	D	**364**/181	494/200	543/206	672/217	724/222

Table 7: Performance comparison between different algorithms with timeouts from 100 to 7800 seconds. All results have the format TΣ/T#. (TΣ) and (T#) mean sort the cascading priority according to TΣ and T#, respectively. (ub) means filtering the configuration set. TS and CA label the methods according to the criteria proposed by Schede et al. (2022): TS: training setting (O: offline, R: real-time); CA: configuration adjustment (D: dynamic, S: static).

real-time to the current tree decomposition. A pseudocode of *ICP-dyn* is shown in Algorithm 1 (*ICP-sta* follows as a more straightforward special case).

We also consider a variant of *ICP-sta*, labeled (ub), which operates with cascades formed by a subset of the 30 configurations. If we can cover the configuration space with fewer best-performing configurations, we can pass through each cascade quicker and consequently carry out more rounds and update the ordering more often. Accordingly, we remove all configurations with "unique best number" $\mathsf{ub} = 0$, meaning these configurations cannot achieve the sole-best performance on an training instances. After filtering, there are 20 configurations left to form a cascade.

Experimental Analysis

With different settings, we have a series of variants for *ICP*. We examine these variants along 5 global timeouts for a fair comparison, and we compare *ICP* with the algorithms proposed above (*CC-sta*, *CC-dyn*, *SB-one*, *SB-all*, and *CA*).

Table 7 presents the overall results of different algorithms and their key properties of techniques. Both static and dynamic *ICPs* outperform other algorithms significantly in all given timeouts: *ICP-sta* (TΣ) obtain 228 and 141 more total improvements than the variants of SMAC (*SB-all* and *SB-one* within 7800 seconds). Moreover, *ICP* surpasses the portfolio-based algorithm selector AutoFolio (*CA*) by 40% TΣ. When comparing *ICP-sta* and *ICP-dyn*, we observe within the timeout range $(100, 1000)$, that *ICP-dyn* (T#) is slightly better than *ICP-stat* (TΣ), but for other timeouts, the static ICP is better. For *ICP-sta*, sorting the cascading priority queue by TΣ is better, which can result in a higher TΣ. In contrast, *ICP-dyn* prefers to have T# as the metric for formulating the priority queues. When it comes to the effect of the configuration filter according to ub, the filter can even improve the performance further within 100 seconds timeout. *ICP* with the filter can still achieve a similar performance level with long timeout: with 7800 seconds, the performance of *ICP-sta* (TΣ) and *ICP-sta* (TΣ) (*ub*) is similar, even though the latter one has fewer configurations in the cascading priority queue.

In general, even though offline algorithm configurators dominate the research domain, we discover that offline configurators are not good at configuring the process where the instance is constantly changing, like in TW-SLIM. In our dynamic context, we can boost the performance further by incorporating knowledge learned during offline configuration (30 configurations in our *CC*), adaptive selection methods (*CC-dyn*), and cascading methods (*ICP*).

Conclusions and Future Work

We have investigated automatically configuring the complex algorithmic framework TW-SLIM for treewidth minimization, which, at its core, uses a SAT solver locally for a large-scale optimization problem. We adapted clustering-based automated algorithm configuration to our highly dynamic setting, which allowed us to improve significantly over the original hand-tuned configuration. Here, we observed that selecting the configuration through clustering performed better than through an algorithm selector. This finding is interesting as it contrasts the results obtained in a different setting on the configuration of MaxSAT solvers (Kadioglu et al. 2010).

Our new iterative cascading policy (ICP) provides a significant additional boost in performance gain. This remarkable performance is due to ICP's ability to self-refine in real-time, learning from one round to the other and simultaneously adapting to a dynamic change in the instance.

Our results give a good picture of the potential of automated algorithm configuration for a complex algorithmic framework that includes multiple calls to a SAT solver. Although we consider a particular optimization problem as our concrete target (MINTW), we are confident that our findings are relevant to many other problems that can be tackled with SLIM, LNS, or other frameworks that utilize multiple SAT calls.

Acknowledgments

The project leading to this publication has received funding from the European Union's Horizon 2020 research and innovation programme under the Maria Skłodowska-Curie grant agreement No. 101034440, the Austrian Science Fund (projects P36420 and P36688), and the Vienna Science and Technology Fund (project ICT19-065).

References

Abseher, M.; Musliu, N.; and Woltran, S. 2017. HTD - A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In Salvagnin, D.; and Lombardi, M., eds., *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, 376–386. Springer Verlag.

Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In Gent, I. P., ed., *Principles and Practice of Constraint Programming - CP 2009*, 142–157. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-04244-7.

Arnborg, S.; Corneil, D. G.; and Proskurowski, A. 1987. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Algebraic Discrete Methods*, 8(2): 277–284.

Audemard, G.; and Simon, L. 2018. On the Glucose SAT Solver. *International Journal on Artificial Intelligence Tools*, 27(1): 1840001:1–1840001:25.

Bannach, M.; Berndt, S.; and Ehlers, T. 2017. Jdrasil: A Modular Library for Computing Tree Decompositions. In Iliopoulos, C. S.; Pissis, S. P.; Puglisi, S. J.; and Raman, R., eds., *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Bodlaender, H. L. 1993. A Tourist Guide through Treewidth. *Acta Cybernetica*, 11: 1–21.

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

Bradley, P. S.; Bennett, K. P.; and Demiriz, A. 2000. Constrained K-means Clustering. *Microsoft Research, Redmond*, 20(0): 0.

Eiben, E.; Ganian, R.; Kanj, I.; and Szeider, S. 2019. The Parameterized Complexity of Cascading Portfolio Scheduling. In Wallach, H. M.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E. B.; and Garnett, R., eds., *Proceedings of NeurIPS 2019, the Thirty-third Conference on Neural Information Processing Systems*, 7666–7676.

Fichte, J. K.; Berre, D. L.; Hecher, M.; and Szeider, S. 2023. The Silent (R)evolution of SAT. *Communications of the ACM*, 66(6): 64–72.

Fichte, J. K.; Lodha, N.; and Szeider, S. 2017. SAT-Based Local Improvement for Finding Tree Decompositions of Small Width. In Gaspers, S.; and Walsh, T., eds., *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, 401–411. Springer Verlag.

Fitzgerald, T.; Malitsky, Y.; and O'Sullivan, B. 2015. ReACTR: Realtime Algorithm Configuration through Tournament Rankings. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*. AAAI.

Fitzgerald, T.; Malitsky, Y.; O'Sullivan, B.; and Tierney, K. 2014. ReACT: Real-time Algorithm Configuration through Tournaments. In *Proceedings of the International Symposium on Combinatorial Search*, volume 5(1), 62–70.

Hinton, G. E.; and Roweis, S. 2002. Stochastic Neighbor Embedding. *Advances in Neural Information Processing Systems*, 15.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential Model-based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, 507–523. Springer.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC - Instance-Specific Algorithm Configuration. In Coelho, H.; Studer, R.; and Wooldridge, M. J., eds., *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, 751–756. IOS Press.

Kloks, T. 1996. Treewidth of circle graphs. *International Journal of Foundations of Computer Science*, 7: 111–120.

Kulikov, A. S.; Pechenev, D.; and Slezkin, N. 2022. SAT-Based Circuit Local Improvement. In Szeider, S.; Ganian, R.; and Silva, A., eds., *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria*, volume 241 of *LIPIcs*, 67:1–67:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Lindauer, M.; Eggensperger, K.; Feurer, M.; Biedenkapp, A.; Deng, D.; Benjamins, C.; Ruhkopf, T.; Sass, R.; and Hutter, F. 2021. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. In *ArXiv: 2109.09831*.

Lindauer, M.; Hoos, H. H.; Hutter, F.; and Schaub, T. 2015. Autofolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53: 745–778.

Lodha, N.; Ordyniak, S.; and Szeider, S. 2016. A SAT Approach to Branchwidth. In Creignou, N.; and Berre, D. L., eds., *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, 179–195. Springer Verlag.

Lodha, N.; Ordyniak, S.; and Szeider, S. 2017. SAT-Encodings for Special Treewidth and Pathwidth. In Gaspers, S.; and Walsh, T., eds., *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, 429–445. Springer Verlag.

López-Ibáñez, M.; Dubois-Lacoste, J.; Pérez Cáceres, L.; Birattari, M.; and Stützle, T. 2016. The Irace Package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives*, 3: 43–58.

Peruvemba Ramaswamy, V.; and Szeider, S. 2021a. Learning Fast-Inference Bayesian Networks. *Advances in Neural Information Processing Systems*, 34.

Peruvemba Ramaswamy, V.; and Szeider, S. 2021b. Turbocharging Treewidth-Bounded Bayesian Network Structure Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5): 3895–3903.

Pisinger, D.; and Ropke, S. 2010. Large neighborhood search. In *Handbook of Metaheuristics*, 399–419. Springer Verlag.

Ramaswamy, V. P.; and Szeider, S. 2020. MaxSAT-Based Postprocessing for Treedepth. In Simonis, H., ed., *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, 478–495. Springer.

Ramaswamy, V. P.; and Szeider, S. 2022. Learning Large Bayesian Networks with Expert Constraints. In Cussens, J.; and Zhang, K., eds., *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*, volume 180 of *Proceedings of Machine Learning Research*, 1592–1601. PMLR.

Reichl, F.; Slivovsky, F.; and Szeider, S. 2023. Circuit Minimization with QBF-Based Exact Synthesis. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*, 4087–4094. AAAI Press.

Roussel, O. 2012. Description of ppfolio 2012. In A. Balint, e. a., ed., *Proceedings of SAT Challenge 2012*, 47. University of Helsinki.

Samer, M.; and Veith, H. 2009. Encoding Treewidth into SAT. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, 45–50. Springer Verlag.

Schede, E.; Brandt, J.; Tornede, A.; Wever, M.; Bengs, V.; Hüllermeier, E.; and Tierney, K. 2022. A Survey of Methods for Automated Algorithm Configuration. *Journal of Artificial Intelligence Research*, 75: 425–487.

Schidler, A. 2022. SAT-Based Local Search for Plane Subgraph Partitions (CG Challenge). In Goaoc, X.; and Kerber, M., eds., *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, 74:1–74:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Schidler, A.; and Szeider, S. 2021. SAT-based Decision Tree Learning for Large Data Sets. In *Proceedings of AAAI'21, the Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press.

Streeter, M. 2018. Approximation Algorithms for Cascading Prediction Models. In Dy, J. G.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *JMLR Workshop and Conference Proceedings*, 4759–4767. JMLR.org.

Tamaki, H. 2022. Heuristic Computation of Exact Treewidth. In Schulz, C.; and Uçar, B., eds., *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Xia, H.; and Szeider, S. 2023. SAT-Based Tree Decomposition with Iterative Cascading Policy Selection. Zenodo.org Online Repository. DOI 10.5281/zenodo.10407175, https://zenodo.org/records/10407175.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32: 565–606.