SAT-Based Local Improvement for Finding Tree Decompositions of Small Width

Johannes K. Fichte✉, Neha Lodha, and Stefan Szeider

TU Wien, Vienna, Austria
jfichte@dbai.tuwien.ac.at, {sz,neha}@ac.tuwien.ac.at

Abstract. Many hard problems can be solved efficiently for problem instances that can be decomposed by tree decompositions of small width. In particular for problems beyond NP, such as #P-complete counting problems, tree decomposition-based methods are particularly attractive. However, finding an optimal tree decomposition is itself an NP-hard problem. Existing methods for finding tree decompositions of small width either (a) yield optimal tree decompositions but are applicable only to small instances or (b) are based on greedy heuristics which often yield tree decompositions that are far from optimal. In this paper, we propose a new method that combines (a) and (b), where a heuristically obtained tree decomposition is improved locally by means of a SAT encoding. We provide an experimental evaluation of our new method.

1 Introduction

Treewidth is arguably the most prominent graph invariant with important application in discrete algorithms and optimization [5,8], constraint satisfaction [11,16], knowledge representation and reasoning [19], computational biology [30], and probabilistic networks and inference [10,24,26]. Treewidth was introduced by Robertson and Seymour in their Graph Minors Project and according to Google Scholar[1], the term is mentioned in over 17000 research articles.

Small treewidth of a graph indicates in a certain sense its tree-likeness and sparsity. Many otherwise NP-hard graph problems such as Hamiltonicity and 3-colorability, but also problems "beyond NP" such as the #P-complete problem of determining the number of perfect matchings in a graph are solvable in polynomial time for graphs of bounded treewidth [9]. Treewidth is based on certain decompositions of graphs, called tree decompositions, where sets of vertices of the input graph are arranged in bags at the nodes of a tree such that certain conditions are satisfied. The width of a tree decomposition is the size of a largest bag minus 1. A tree decomposition is optimal for a given graph if the graph has no tree decomposition of smaller width. The treewidth of a graph is the width of an optimal tree decomposition.

Algorithms that exploit the small treewidth of a graph usually proceed by dynamic programming along the tree decomposition where at each node of the tree, information

Research was supported by the Austrian Science Fund (FWF), Grants Y698, W1255-N23, and P-26200. The first author is also affiliated with the University of Potsdam, Germany.

[1] Retrieved on March 26, 2017.

is gathered in tables. The size of these tables is usually exponential or even double exponential in the size of the bag. Thus, it is important to obtain a tree decomposition of small width. However, since finding an optimal tree decomposition is an NP-hard task [2], the following two main approaches have been proposed in the literature:

(a) *Exact methods* that compute optimal tree decompositions. Optimal tree decompositions are found using specialized combinatorial algorithms based on graph separators [2], branch-and-bound algorithms [18], but also by means of *SAT encodings* [4,28]. These exact methods are limited to rather small graphs with about hundred vertices.
(b) *Heuristic methods* that compute sub-optimal tree decompositions. These algorithms are usually based on so-called elimination orderings which are found by a greedy approach [6,20]. The heuristic methods are quite fast and scale up to large graphs with thousands of vertices, but lead to tree decompositions that can be far from optimal.

In fact, because of the split into these two categories of algorithmic approaches, also the recent PACE challenge [13], where finding good tree decompositions was one of the main tasks, featured two respective categories: one asking for the exact treewidth of small graphs, and one asking for sub-optimal tree decompositions of large graphs.

SAT-Based Local Improvement In this paper, we propose a new approach to finding tree decompositions, which combines exact methods with heuristics. The basic idea is to (i) start with a tree decomposition obtained with a heuristic method (the *global solver*) and (ii) subsequently select parts of the tree decomposition, trying to improve it with another method (the *local solver*). It turned out that SAT-based exact methods are particularly well-suited for providing the local solver.

Consider a given graph G and a tree decomposition $\mathcal{T}$ of G, obtained by the global solver. We select a small part $\mathcal{S}$ of $\mathcal{T}$, which is a tree decomposition of the subgraph $G_{\mathcal{S}}$ of G, induced by all the vertices that appear in bags at nodes in $\mathcal{S}$. Once the local solver finds a better tree decomposition of $G_{\mathcal{S}}$, we would like to replace $\mathcal{S}$ in $\mathcal{T}$ with the new tree decomposition found by the local solver. This, however, does not work in general, as the new tree decomposition might not fit into the remaining parts of $\mathcal{T}$. Fortunately we can make this approach work by using the following trick. We add to $G_{\mathcal{S}}$ certain cliques, which we call *marker cliques*, and which tell us how to replace the original local tree decomposition $\mathcal{S}$ with the new one. Due to a general property of tree decompositions, there is always a bag that contains all vertices of a clique. Hence, in particular, the new local decomposition will contain for each marker clique a bag that contains it, and this bag will be an anchor point for connecting the new decomposition to the parts of the old one. Details of this construction are explained in Section 3.

Related Work A SAT-based local improvement approach was first proposed, implemented and evaluated by Lodha et al. [25] for finding *branch decompositions* of small width of graphs and hypergraphs. As the definitions of a branch decompositions and tree decompositions differ significantly, the methods for finding and replacing local decompositions are quite different. Also the SAT encoding of branchwidth and treewidth are different, as the former focuses on edges, while the latter focuses on vertices.

There are several approaches for improving treewidth heuristics based on elimination orderings. For instance, Kask et al. [21] use randomization to recompute the last few steps of the ordering computed so far, picking the best of the runs, whereas Gaspers et al. [17] use a different approach: as soon as a given width bound is exceeded during the computation of the ordering, the last c vertices of the ordering are recomputed with an exact method, trying to stay within the width bound.

2 Preliminaries

In this section we introduce some relevant graph theoretic notions.

All considered graphs are finite, simple, and undirected. Let G be a graph. $V(G)$ and $E(G)$ denote the vertex set and the edge set of G, respectively. We denote an edge between vertices u and v by uv (or equivalently by vu). The subgraph of G induced by a set $S \subseteq V(G)$ has as vertex set S and as edge set $\{\, uv \in E(G) \mid u, v \in S \,\}$.

A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a tree and χ is a mapping that assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called a *bag*, such that the following conditions hold (we refer to the vertices of T as *nodes* to make the distinction between T and G clearer).

1. $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq \bigcup_{t \in V(T)} \{\, uv \mid u, v \in \chi(t) \,\}$.
2. The sets $\chi(t_1) \setminus \chi(t)$ and $\chi(t_2) \setminus \chi(t)$ are disjoint for any three nodes $t, t_1, t_2 \in V(T)$ such that t lies on a path from t_1 to t_2 in T.

The *width* of $\mathcal{T}$, denoted $\mathsf{w}(\mathcal{T})$, is $\max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* $\mathsf{tw}(G)$ of G is the minimum $\mathsf{w}(\mathcal{T})$ over all tree decompositions $\mathcal{T}$ of G.

We will make use of the following well-known fact.

Fact 1 ([23]) *Let (T, χ) be a tree decomposition of a graph G and K a clique in G. Then there exists at least one node $t \in V(T)$ such that $V(K) \subseteq \chi(t)$.*

3 Local Improvement of Tree Decompositions

3.1 Local Tree Decompositions

Let G be a graph and $\mathcal{T} = (T, \chi)$ a tree decomposition

For the following considerations we fix a graph G and a tree decomposition $\mathcal{T} = (T, \chi)$ of G. We consider a subtree S of T.

We call $\mathcal{S} = (S, \chi_S)$ a *local tree decomposition of $\mathcal{T}$* (*induced by S*), where χ_S is the restriction of χ to the nodes of S. Let $G_{\mathcal{S}}$ denote the subgraph of G induced by all the vertices of G that appear in a bag of $\mathcal{S}$. The following observation is an immediate consequence of the definitions.

Observation 1 *$\mathcal{S}$ is a tree decomposition of $G_{\mathcal{S}}$ of width $\leq \mathsf{w}(\mathcal{T})$.*

Our goal is to replace $\mathcal{S}$ with an improved tree decomposition $\mathcal{S}'$ of $G_{\mathcal{S}}$, i.e., one of smaller width, and to insert $\mathcal{S}'$ back into $\mathcal{T}$ so that we obtain a new tree decomposition $\mathcal{T}'$ of G of possibly smaller width. In order to make this work, we need to modify $G_{\mathcal{S}}$ such that any tree decomposition of the modified graph can be added back into $\mathcal{T}$.

Let us first introduce some auxiliary notions. For an edge st of $\mathcal{T}$ we define $\lambda_{\mathcal{T}}(st) = \chi(s) \cap \chi(t)$ to the *cut set* associated with st. We call an edge st of T to be a *boundary edge* (w.r.t. $\mathcal{S}$) if $s \in V(S)$ and $t \notin V(S)$.

Now we define the *augmented local graph* $G^*_{\mathcal{S}}$ by setting $V(G^*_{\mathcal{S}})$ to be the set of all vertices of G that appear in a bag of $\mathcal{S}$, and $E(G^*_{\mathcal{S}})$ to be the set of edges uv with $u, v \in V^*$ such that $uv \in E(G)$ or $u, v \in \lambda_{\mathcal{T}}(e)$ for a boundary edge e of T. In other words, the augmented local graph $G^*_{\mathcal{S}}$ is obtained from $G_{\mathcal{S}}$ by forming cliques over cut sets associated with boundary edges. We will use these cliques as "markers" in order to connect a new tree decomposition of $G^*_{\mathcal{S}}$ to the parts of the tree decomposition $\mathcal{T}$ that we keep. Therefore we call these cliques *marker cliques*.

Observation 2 *$\mathcal{S}$ is a tree decomposition of $G^*_{\mathcal{S}}$ of width $\leq \mathsf{w}(\mathcal{T})$.*

Proof. In view of Observation 2 it remains to check that for each edge $uv \in E(G^*_{\mathcal{S}}) \setminus E(G_{\mathcal{S}})$ there is a node s of S such that $u, v \in \chi(s)$. For such an edge uv there is a boundary edge e of T such that $u, v \in \lambda_{\mathcal{T}}(e)$. By definition of a boundary edge, exactly one end of e, say s, belongs to $V(S)$. Now $u, v \in \lambda_{\mathcal{T}}(e) \subseteq \chi(s)$. □

Let $\mathcal{S}^* = (S^*, \chi^*)$ be another tree decomposition of $G^*_{\mathcal{S}}$ with $\mathsf{w}(\mathcal{S}^*) \leq \mathsf{w}(\mathcal{S})$. W.l.o.g., we assume that S^* and T do not share any vertices (if not, we can simply use a tree that is isomorphic to S^*). We define a new tree decomposition $\mathcal{T}' = (T', \chi')$ of G as follows.

Let $T_1, \ldots, T_r$ be the connected components of $T - S$ (each T_i is a tree). Each T_i gives raise to a local tree decomposition $\mathcal{T}_i = (T_i, \chi_i)$ where χ_i is the restriction of χ to the nodes of T_i.

For each T_i let t_i be the leaf of T_i that was incident with a boundary edge $e_i = t_i s_i$ in T. The boundary edge e_i is responsible for a marker clique $K(e_i)$ on the vertices in $\lambda_{\mathcal{T}}(e_i)$. By Fact 1, we can choose a node $s'_i \in V(S^*)$ such that $V(K(e_i)) = \lambda_{\mathcal{T}}(e_i) \subseteq \chi^*(s'_i)$.

We define a new tree decomposition $\mathcal{T}' = (T', \chi')$ where T' is the tree defined by $V(T') = V(S^*) \cup \bigcup_{i=1}^{r} V(T_i) = V(S^*) \cup V(T) \setminus V(S)$ and $E(T') = E(S^*) \cup \bigcup_{i=1}^{r} E(T_i) \cup \{t_1 s'_1, \ldots, t_r s'_r\}$. It remains to define the bags of the tree decomposition $\mathcal{T}'$. For $t \in V(T_i)$ we define $\chi'(t) = \chi(t)$ and for $s \in V(S^*)$ we define $\chi'(s) = \chi^*(s)$.

We denote $\mathcal{T}'$ as $\mathcal{T}\binom{\mathcal{S}}{\mathcal{S}'}$ and say that $\mathcal{T}'$ *is obtained from $\mathcal{T}$ by replacing $\mathcal{S}$ with $\mathcal{S}'$*.

Observation 3 *$\mathcal{T}\binom{\mathcal{S}}{\mathcal{S}'}$ is a tree decomposition of G of width*

$$\max(\mathsf{w}(\mathcal{T}_1), \ldots, \mathsf{w}(\mathcal{T}_r), \mathsf{w}(\mathcal{S}^*)) \leq \max(\mathsf{w}(\mathcal{T}), \mathsf{w}(\mathcal{S}^*)) \leq \max(\mathsf{w}(\mathcal{T}), \mathsf{w}(\mathcal{S})) \leq \mathsf{w}(\mathcal{T}).$$

Proof. Let $\mathcal{T}\binom{\mathcal{S}}{\mathcal{S}'} = \mathcal{T}' = (T'\chi')$. First we observe that T' is indeed a tree, as each tree T_i is connected to the central tree S^* with exactly one edge. Clearly $\mathcal{T}'$ satisfies the first of the two conditions in the definition of a tree decomposition. To see that it also satisfies the second condition, we observe that if a vertex v of G appears in bags at two different local tree decompositions $\mathcal{T}_i$ and $\mathcal{T}_j$ then v must also appear in the

sets $\lambda_{\mathcal{T}}(e_i)$ and $\lambda_{\mathcal{T}}(e_j)$. Consequently, it appears in the bags of s'_i and s'_j (we use the notation from above). As $\mathcal{S}^*$ satisfies the second condition of a tree decomposition, v is contained in all the bags on the path between s'_i and s'_j in S^*. This shows that $\mathcal{T}'$ is indeed a tree decomposition of G. The claimed bound on its width follows directly from the construction. □

3.2 SAT Encodings for Tree Decompositions

A SAT encoding for tree decompositions was first proposed by Samer and Veith [28]. Given a graph G and an integer k, a CNF formula $\Phi(G, k)$ is produced which is satisfiable if and only if G has a tree decomposition of width $\leq k$. For the construction of $\Phi(G, k)$, an alternative characterization of tree decompositions in terms of *elimination orderings* is used. Here a linear ordering of the given graph G is guessed, and based on the ordering certain "fill-in edges" are added to the graph, providing a "triangulation" of G. The ordering is represented by Boolean variables, one for every pair of vertices, whose truth value indicates the relative ordering of the two vertices. Transitivity of the ordering is ensured by suitable clauses. Then, for each vertex v of G it is checked whether it has at most k neighbors that appear in the ordering right to v. This is checked via cardinality constraints [29]. The exact treewidth is then found by systematically calling a SAT solver for a heuristically computed upper bound u with $\Phi(G, k)$ for $k = u, u-1, u-2, \ldots$ and until $\Phi(G, k)$ is found unsatisfiable. From a satisfying assignment of $\Phi(G, k)$ one can obtain a tree decomposition of G of width k efficiently by a *decoding procedure*.

3.3 The Local Improvement Loop

We describe the overall algorithm. Let G be an input graph. First we obtain a tree decomposition $\mathcal{T} = (T, \chi)$ of G using a standard heuristic method, which we refer to as the **global solver**.

The local improvement loop operates with the following parameters which are positive integers: the local budget **lb**, the local timeout **lt**, the global timeout **gt**, and the number of no-improvement rounds **ni**.

We select a node t from T with largest bag size, i.e., $|\chi(t)| = \mathsf{w}(\mathcal{T})$.

In T we perform a modified breadth-first-search (BFS) starting at t. We use an auxiliary set variable L which, at the beginning of the BFS is set to $\chi(t)$. For each node t' visited by the BFS, we add the new elements of $\chi(t')$ to L. If a node t' was visited via an edge e, a neighbor t'' of t' is only visited if $\lambda_{\mathcal{T}}(t't'') < \lambda_{\mathcal{T}}(e)$. The BFS terminates as soon as visiting another node would increase the size of L beyond the local budget **lb**. Now the visited nodes induce a subtree S of T, and in turn, this yields a local tree decomposition $\mathcal{S} = (S, \chi_S)$ of $\mathcal{T}$, as defined above. The set L contains the vertices of the local graph $G_{\mathcal{S}}$ (or equivalently, of the augmented local graph $G^*_{\mathcal{S}}$) which by construction can be at most **lb** many vertices.

Next we run the **local solver**, that is, we check satisfiability of the formula obtained by the SAT encoding, trying to get a tree decomposition $\mathcal{S}^*$ of G^*_S whose width is as small as possible. We start the SAT encoding with $k = \mathsf{w}(\mathcal{S}) - 1$ and upon success decrease k step by step. Each SAT-call has a timeout of **lt** seconds, and we stop if either we get an unsatisfiable instance or we hit the timeout. With the reached value of

k, the treewidth of G_S^* is at most $k+1$. Since the SAT encoding with value $k+1$ is satisfiable, we can extract with a decoding procedure from the satisfying assignment a tree decomposition $\mathcal{S}^*$ of G_S^*. Now we replace $\mathcal{S}$ in $\mathcal{T}$ by $\mathcal{S}^*$, and we repeat the local improvement loop with $\mathcal{T}\binom{\mathcal{S}}{\mathcal{S}^*}$. We note that a local replacement is done even if there was no local width improvement, i.e., if $\mathsf{w}(\mathcal{S}^*) = \mathsf{w}(\mathcal{S})$, as there is the possibility that the change triggers improvements in subsequent rounds of the local improvement loop.

We repeat the local improvement loop until either the global timeout **gt** is reached, or if the loop has been iterated **ni** times without any local width improvements.

4 Experimental Results

Solvers As the global solver we used the greedy ordering heuristics-based algorithm from Abseher et al. [1, rev. 075019f] which we refer to as `heur`. It computes upper bounds for treewidth and outputs a certificate decomposition. The solver scored third in the heuristic track of of the PACE 2016 challenge [13]. It is very space efficient and reports initial useful tree decompositions extremely fast compared to other solvers. It leaves almost the full time resource for the local improvement. We used the following three local solvers:

1. `sat`: a solver based on an improved version of Samer and Veith's [28] SAT encoding by Bannach et al. [3, rev. 25d6a98]. The solver employs Glucose as a SAT solver, PBLib for cardinality encodings, and progresses downwards from an upper bound. The solver scored third in the exact track of the PACE 2016 treewidth challenge and was there the best SAT-based solver.
2. `comb`: an implementation of Arnborg et al.'s combinatorial algorithm [2] by Tamaki [31, rev. d5ba92a], This solver won the exact track of the PACE 2016 treewidth challenge. It incrementally checks for the exact treewidth, it progresses upwards from 1.
3. `heur`: the same solver that we also use as global solver.

Our implementation is publicly available on GitHub [15]. Our experiments mainly focus on two questions: (i) can we improve with local improvement over traditional greedy heuristics and (ii) which solvers are favorable as local solver.

Instances We considered an initial selection of overall 3168 graphs from various publicly available graph sets. Our sets consisted of the *TreewidthLIB* [7], networks from the *UAI competition* [12], publicly available transit graphs from *GTFS-transit feeds* [14], and graphs from the *PACE 2016 treewidth challenge* [13]. Since we aimed for larger graphs where exact methods cannot be used, we restricted ourselves to graphs that contain more than 100 vertices, resulting in 1946 graphs in total.

Experimental Setup The experiments ran on a Scientific Linux cluster of 24 nodes (2x Xeon E5520 each) and overall 224 physical cores [22]. Due to the large number of instances, we started only from one initial decomposition (with random seed) and did not repeat the runs. In order to have reproducible results we used a benchmark cluster

Table 1. Summary of treewidth improvements.

#improved	improvements (sum)	improvement (max)	solver configuration
647	**2015**	13	`sat-100-1800(900)`
584	1984	16	`sat-125-1800(900)`
630	1805	15	`comb-100-1800`
493	1676	**20**	`sat-150-1800(900)`
631	1548	12	`comb-075-1800`
609	1460	12	`comb-075-1800`
447	1077	19	`comb-125-1800`
368	822	14	`comb-150-1800`
325	538	9	`heur-150-1800`
258	421	8	`heur-100-1800`

run generator and analysis tool[2]. All solvers have been compiled with gcc version 4.9.1, ran on Python 2.7.5, and Java 1.8.0_122 HotSpot 64-bit server VM, respectively. We executed solvers in single core mode. We limited available memory (RAM) to 8GB, wall clock time of the global solver to 15 seconds, wall clock time of the overall search to 7800 seconds, and wall clock time of the local solver to 1800 seconds. For the SAT solver we imposed an additional restriction that the individual SAT call runs at most 900 seconds (**st**). Resource limits where enforced by *runsolver* [27].

For our experiments, we systematically tested the parameters **lb** $\in \{75, 100, 125, 150\}$, **lt** $\in \{90, 900, 1800\}$, **gt** $= 7200$, and **ni** $= 10$. For the parameter **ni** we also tried values 40 and 100 on a selected set of instances, but obtained no improvements. Individual results are publicly available [15].

Results Table 1 summarizes the improvements we obtained with our experiments. Configurations in the legend are given in the form `solver-lb-lt(st)`. The best results in each column are highlighted in bold font. Table 2 shows some of the best and notable improvements we obtained with local improvements. The value "hash" provides the first four digits of sha-1 hash sum for the instance in DIMACS graph format. Column "htw" has the heuristically obtained treewidth, and "itw" has the treewidth after local improvement. The configuration with which we got these improvements are in the column "local solver." The best improvement we obtained is 20, for the instance or_chain_224.fg, from the graph set networks. Among further entries in the table are instance graph13pp with a width over 100, and instance Promedus_38 where we could reduce the width from 23 to 16, which makes this instance feasible for dynamic programming.

Discussion For our instance set, we can see that even a heuristic solver as local solver (**lb** $= 150$) improved the upper bounds. Both in terms of number of improved instances and when considering the cumulative sum of improvements, the SAT-based solver

[2] The run and analysis tool is available online at https://github.com/daajoe/benchmark-tool. The file benchmark-tool/runscripts/treewidth/localimprovement.xml contains all solver flags to reproduce our benchmark runs.

Table 2. Some of the best and notable improvements

instance (hash)	$\|V\|$	$\|E\|$	graphs	itw	htw	local solver
or_chain_224.fg (a4cb)	1638	3255	networks	75	95	`sat-150-1800-10`
or_chain_54.fg (a6fc)	1404	2757	networks	65	84	`comb-125-1800-10`
or_chain_187.fg (826a)	1668	3197	networks	79	97	`sat-150-1800-10`
1bkr_graph (003a)	107	1340	twlib	44	56	`comb-075-1800-10`
dimacs_fpsol2.i.1-pp (69aa)	191	4418	pace2016	61	72	`sat-150-1800-10`
graph13pp (eb9d)	456	1874	twlib	115	125	`comb-150-1800-10`
Cell120 (b625)	600	1200	pace2016	94	104	`comb-150-1800-10`
bkv-zrt_20120422_0314 (fbca)	907	2209	transit	74	83	`sat-150-1800-10`
Promedus_38 (02d7)	668	1235	networks	16	23	`sat-150-1800-10`

performed best. For both the combinatorial solver and the SAT-based solver, a local budget $\mathbf{lb} = 100$ resulted in more solved instances. However, in terms of overall improvement the difference between the two local solvers is small. A local budget $\mathbf{lb} = 125$ allowed us to increase the cumulative sum of improvements relatively early.

In consequence, we obtained the best results by using a SAT-based solver as local solver. Using a SAT-based solver, we can hope that an improved SAT encoding or new techniques in solvers immediately yield better upper bounds for treewidth using local improvement. We also computed the virtually best solver, which improved 200 instances more than the best SAT-based configuration. This indicates that we can very likely improve a much higher number of instances when applying a portfolio based solving approach.

5 Concluding Remarks

We have presented a new SAT-based approach to finding tree decompositions of small width based on a cross-over between standard heuristic methods and exact methods. Our work offers several directions for further research.

For instance, one could possibly improve the current setup by (a) upgrading the method for selecting the local tree decomposition, which is currently based on a relatively simple breadth-first-search, and (b) tuning and optimizing the SAT-based local solver specially to handle the type of instances that arise within the local improvement loop.

Another promising direction involves adding additional constraints to the SAT encoding, which yield local tree decompositions with special properties. For instance, when the local solver cannot improve the width of the current local tree decomposition, it could still replace it with one that increases the likelihood of success for further rounds of local improvements (for instance, by minimizing the number of large bags). Another application would be the computation of "customized tree decompositions" [1] which are designed to speed-up dynamic programming algorithms. Such additional constraints are relatively easy to build into a SAT-based local solver, but seem difficult to build into a local solver based on combinatorial methods.

Finally, due to the modularity of our approach (local solver, budget, time out, invoked SAT solver), it could benefit from automated algorithm configuration and parameter tuning, and it could provide the elements of a portfolio approach.

References

1. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'17) (2017)
2. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods 8(2), 277–284 (1987)
3. Bannach, M., Berndt, S., Ehlers, T.: Jdrasil: A modular library for computing tree decompositions. Tech. rep., Lübeck University, Germany (2016)
4. Berg, J., Järvisalo, M.: SAT-based approaches to treewidth computation: An evaluation. In: Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'14. pp. 328–335. IEEE Computer Soc., Limassol, Cyprus (Nov 2014)
5. Bodlaender, H.L., Koster, A.M.C.A.: Combinatorial optimization on graphs of bounded treewidth. Comput. J. 51(3), 255–269 (2008)
6. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations. I. Upper bounds. Information and Computation 208(3), 259–275 (2010)
7. van den Broek, J.W., Bodlaender, H.: TreewidthLIB – a benchmark for algorithms for treewidth and related graph problems. Tech. rep., Faculty of Science, Utrecht University (2010), http://www.staff.science.uu.nl/~bodla101/treewidthlib/
8. Chimani, M., Mutzel, P., Zey, B.: Improved Steiner tree algorithms for bounded treewidth. J. Discrete Algorithms 16, 67–78 (2012)
9. Courcelle, B., Makowsky, J.A., Rotics, U.: On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. Discr. Appl. Math. 108(1-2), 23–52 (2001)
10. Darwiche, A.: A differential approach to inference in Bayesian networks. J. ACM 50(3), 280–305 (2003)
11. Dechter, R.: Tractable structures for constraint satisfaction problems. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, vol. I, chap. 7, pp. 209–244. Elsevier (2006)
12. Dechter, R.: Graphical model algorithms at UC Irvine. Tech. rep., UC Irvine (2013), http://graphmod.ics.uci.edu/group. The network instances consist of Bayesian and Markov networks used in UAI competition and protein folding/side-chain prediction problems.
13. Dell, H., Rosamond, F.: The parameterized algorithms and computational experiments challenge. https://pacechallenge.wordpress.com/ (2016)
14. Fichte, J.K.: daajoe/gtfs2graphs – a GTFS transit feed to graph format converter. https://github.com/daajoe/gtfs2graphs (2016)
15. Fichte, J.K., Lodha, N., Szeider, S.: trellis: Treewidth local improvement solver. https://github.com/daajoe/trellis (2017)
16. Freuder, E.C.: A sufficient condition for backtrack-bounded search. J. ACM 32(4), 755–761 (1985)
17. Gaspers, S., Gudmundsson, J., Jones, M., Mestre, J., Rümmele, S.: Turbocharging Treewidth Heuristics. In: Guo, J., Hermelin, D. (eds.) 11th International Symposium on Parameterized and Exact Computation (IPEC 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 63, pp. 13:1–13:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017)

18. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: Proceedings of the Proceedings of the Twentieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-04). pp. 201–208. AUAI Press, Arlington, Virginia (2004)
19. Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. Artificial Intelligence 174(1), 105–132 (2010)
20. Hammerl, T., Musliu, N., Schafhauser, W.: Metaheuristic algorithms and tree decomposition. In: Kacprzyk, J., Pedrycz, W. (eds.) Springer Handbook of Computational Intelligence, pp. 1255–1270. Springer Verlag, Berlin, Heidelberg (2015)
21. Kask, K., Gelfand, A., Otten, L., Dechter, R.: 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. AAAI Press (2011)
22. Kittan, K.: Zuse cluster. http://www.cs.uni-potsdam.de/bs/research/labsZuse.html (2017)
23. Kloks, T.: Treewidth: Computations and Approximations. Springer Verlag, Berlin (1994)
24. Lauritzen, S.L., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems. J. Roy. Statist. Soc. Ser. B 50(2), 157–224 (1988)
25. Lodha, N., Ordyniak, S., Szeider, S.: A SAT approach to branchwidth. In: Creignou, N., Berre, D.L. (eds.) Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing, SAT 2016. Lecture Notes in Computer Science, vol. 9710, pp. 179–195. Springer Verlag (2016)
26. Ordyniak, S., Szeider, S.: Parameterized complexity results for exact Bayesian network structure learning. J. Artif. Intell. Res. 46, 263–302 (2013)
27. Roussel, O.: Controlling a solver execution with the runsolver tool. J on Satisfiability, Boolean Modeling and Computation 7, 139–144 (2011)
28. Samer, M., Veith, H.: Encoding treewidth into SAT. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009. Lecture Notes in Computer Science, vol. 5584, pp. 45–50. Springer Verlag (2009)
29. Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints. In: van Beek, P. (ed.) Proceedings of the 11th International Conference Principles and Practice of Constraint Programming, CP 2005. Lecture Notes in Computer Science, vol. 3709, pp. 827–831. Springer Verlag (2005)
30. Song, Y., Liu, C., Malmberg, R.L., Pan, F., Cai, L.: Tree decomposition based fast search of RNA structures including pseudoknots in genomes. In: Proceedings of the 4th International IEEE Computer Society Computational Systems Bioinformatics Conference, CSB 2005. pp. 223–234. IEEE Computer Society (2005)
31. Tamaki, H.: Tcs-meiji. https://github.com/TCS-Meiji/treewidth-exact (2016)