Encoding Treewidth into SAT

Marko Samer and Helmut Veith

Department of Computer Science
TU Darmstadt, Germany
{samer,veith}@cs.tu-darmstadt.de

Abstract. One of the most important structural parameters of graphs is *treewidth*, a measure for the "tree-likeness" and thus in many cases an indicator for the hardness of problem instances. The smaller the treewidth, the closer the graph is to a tree and the more efficiently the underlying instance often can be solved. However, computing the treewidth of a graph is NP-hard in general. In this paper we propose an encoding of the decision problem whether the treewidth of a given graph is at most k into the propositional satisfiability problem. The resulting SAT instance can then be fed to a SAT solver. In this way we are able to improve the known bounds on the treewidth of several benchmark graphs from the literature.

1 Introduction

Many important combinatorial problems that are NP-hard in general are easy to solve if the instance is structured as a tree. *Tree decompositions* and the corresponding measure *treewidth* [13] gradually generalize the tree structure of instances to "tree-likeness". Roughly speaking, the smaller the treewidth, the less cyclic the graph and the closer the graph is to a tree. Instances with small treewidth can often be solved efficiently by dynamic programming on the tree decomposition. For example, Courcelle's Theorem [9] states that fixed graph properties expressible in monadic second-order logic (MSO) can be decided in linear time on graphs with bounded treewidth. However, for unbounded k, deciding whether a graph has treewidth at most k is NP-complete [1]. For this reason, the common way in practice for computing tree decompositions is the use of heuristics and approximation algorithms [2,4,8,12]. Although these methods often lead to tree decompositions of small width, the so obtained results are in general only upper bounds on the treewidth: the actual treewidth is the minimum width over all possible tree decompositions. Complementary to these upper bound methods, several approaches for computing lower bounds have been proposed as well [4,5,6,8,12]. Clearly, the treewidth of a graph has been found if the lower and upper bounds coincide. For many graphs, however, the lower and upper bound algorithms yield only an interval in which the treewidth is contained. To make these intervals smaller or to determine the treewidth, also exact algorithms based on branch-and-bound have been developed [3,11].

In this paper we propose one more algorithmic tool for determining the treewidth of graphs: We present an encoding of the decision problem whether a given graph $G = (V, E)$ has treewidth at most k to an instance F of the propositional satisfiability problem (SAT) such that G has treewidth less than or equal to k if and only if F is satisfiable. In this way we aim at exploiting the increasing power of modern SAT solvers to find

O. Kullmann (Ed.): SAT 2009, LNCS 5584, pp. 45–50, 2009.
© Springer-Verlag Berlin Heidelberg 2009

tree decompositions of minimal width and thus to determine the treewidth of graphs. Our encoding results in SAT instances consisting of $\mathcal{O}(k\,|V|^2)$ variables and $\mathcal{O}(|V|^3)$ clauses; the length of each clause is bounded by 4. Using this approach, we have been able to determine the treewidth (or at least to shorten its interval) of several moderately sized benchmark graphs from the literature.

This paper is organized as follows: In Section 2, we formally introduce the notion of tree decompositions and treewidth. Moreover, we explain an alternative characterization of treewidth on which our encoding is based. Then, in Section 3, we present our encoding into propositional satisfiability and our experimental results. Finally, we conclude in Section 4.

2 Treewidth

A *graph* is a tuple (V, E) of a non-empty set V of vertices and a set $E \subseteq \{\{x, y\}\, x, y \in V\}$ of edges. A *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (V', E')$ where each node $t \in V'$ is associated with a "bag" $\chi(t) \subseteq V$ containing vertices of G such that the following conditions are satisfied [13]:

1. $\bigcup_{t \in V'} \chi(t) = V$.
2. For every edge $e \in E$, there is some node $t \in V'$ such that $e \subseteq \chi(t)$.
3. For all nodes $t_1, t_2, t_3 \in V'$ such that t_2 lies on the (unique) path from t_1 to t_3 in T, it holds that $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$.

The first condition requires that there are only vertices in the bags and every vertex occurs in some bag, the second condition requires that all pairs of vertices that are connected by an edge occur together in some bag, and the third condition ("connectedness condition") requires that if a vertex occurs in the bags of two different nodes, then it must occur in each bag on the unique path between them. The *width* of a tree decomposition $T = (V', E')$ is given by $\max_{t \in V'} |\chi(t)| - 1$. The *treewidth* of a graph is the minimum width over all its tree decompositions.

It is well known that from every linear ordering of the vertices of a graph $G = (V, E)$ one can easily construct a tree decomposition of G and that there is always a linear ordering such that the corresponding tree decomposition is optimal, i.e., its width equals the treewidth of G (see, e.g., Bodlaender [4] or Dechter [10]). Our encoding will be based on this alternative characterization. In fact, the linear ordering gives rise to a triangulation of the graph from which the corresponding width can be directly read off, i.e., we do not need to construct the tree decomposition explicitly. In this context, the treewidth is also called *induced width* [10]. Given a linear ordering $v_1, v_2, \ldots, v_n$ of the vertices in V, we call v_j a *predecessor* of v_i if $\{v_i, v_j\} \in E$ and $j < i$, and we call v_j a *successor* of v_i if $\{v_i, v_j\} \in E$ and $j > i$. To determine the width of the tree decomposition obtained from this linear ordering, we proceed as follows: For each pair of non-adjacent vertices v_i and v_j, we successively add an edge $\{v_i, v_j\}$ to E if and only if v_i and v_j have a common predecessor. This is repeated as long as new edges can be added. The width of the tree decomposition is then the maximum number of successors over all vertices, i.e., $\max_{v_i \in V} |\{\{v_i, v_j\} \in E : j > i\}|$. Figure 1 shows a

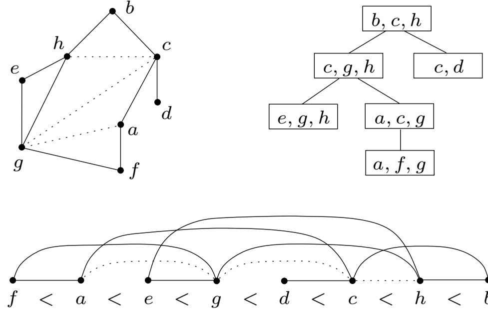

Fig. 1. A graph of treewidth 2, one of its tree decompositions of width 2, and a corresponding linear ordering (the dotted edges are added depending on the ordering)

graph, one of its tree decompositions, and the corresponding linear ordering. The solid edges belong to the input graph and the dotted edges are added according to the above rule: The edge $\{a, g\}$ is added because f is a predecessor of both a and g. This newly added edge now causes the edge $\{c, g\}$ to be added, which in turn causes the edge $\{c, h\}$ to be added. Since the maximum number of successors after this process is 2, we know that the corresponding tree decomposition has width 2.

3 The Encoding

Our encoding into propositional satisfiability is now based on the linear ordering as described in Section 2. Let $G = (V, E)$ be the input graph with $n = |V|$ and $m = |E|$ and assume that all vertices are enumerated from 1 to n. To encode the linear ordering itself, we introduce $n(n-1)/2$ Boolean variables $ord_{i,j}$ with $1 \leq i < j \leq n$: Variable $ord_{i,j}$ is true if and only if vertex v_i precedes vertex v_j in the linear ordering. In addition, we need $n(n-1)(n-2)$ clauses of the form

$$(ord^*_{i,j} \land ord^*_{j,l}) \rightarrow ord^*_{i,l}, \text{ where } ord^*_{p,q} = \begin{cases} ord_{p,q} & \text{if } p < q \\ \neg ord_{q,p} & \text{otherwise} \end{cases}$$

to enforce transitivity. Note that $ord^*_{p,q}$ is used here for presentation purposes only and is replaced in our encoding by $ord_{p,q}$ and $\neg ord_{q,p}$, respectively.

Having now encoded the linear ordering, we are going to encode the successor relation induced by this ordering. To this aim, note that the linear ordering gives a direction to all edges $\{v_i, v_j\} \in E$, namely from v_i to v_j if and only if v_j is a successor of v_i. Thus, we introduce n^2 Boolean variables $arc_{i,j}$ with $1 \leq i, j \leq n$: Variable $arc_{i,j}$ is true if vertex v_j is a successor of vertex v_i. We call such an arc variable *active* if it is assigned true and *inactive* otherwise. As mentioned in Section 2, the maximum number of successors, i.e., the maximum number of active outgoing arcs, is then the width of the corresponding tree decomposition. That means for each vertex we have to ensure by a cardinality constraint that the number of its active outgoing arcs does not exceed

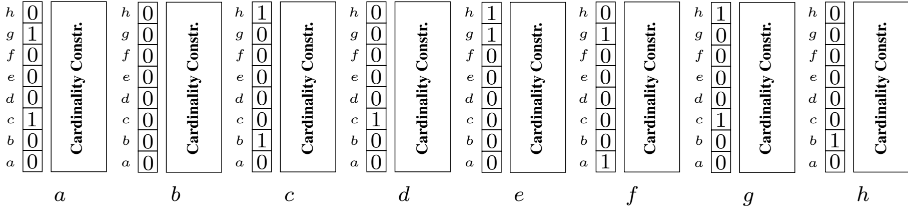

Fig. 2. Encoding of the successor relation induced by the linear ordering in Figure 1

our upper bound k on the treewidth. Figure 2 illustrates this encoding of the successor relation induced by the linear ordering in Figure 1.

Let us consider these steps in more detail. To encode the edges of the input graph, we add $2m$ clauses of the form $ord_{i,j} \rightarrow arc_{i,j}$ and $\neg ord_{i,j} \rightarrow arc_{j,i}$ for $\{v_i, v_j\} \in E$ and $i < j$. Moreover, in order to enforce the additional edges caused by the linear ordering (i.e., the dotted edges in Figure 1), we add $n(n-1)(n-2)$ clauses of the form

$$(arc_{i,j} \wedge arc_{i,l} \wedge ord_{j,l}) \rightarrow arc_{j,l} \quad \text{and} \quad (arc_{i,j} \wedge arc_{i,l} \wedge \neg ord_{j,l}) \rightarrow arc_{l,j}$$

for $1 \leq i, j, l \leq n$, $i \neq j$, $i \neq l$, and $j < l$. We also add $n(n-1)(n-2)/2$ clauses of the form $\neg arc_{i,j} \vee \neg arc_{i,l} \vee arc_{j,l} \vee arc_{l,j}$, which are actually redundant but accelerate the SAT solving process. In addition, since self loops can be neglected, we add n unit clauses of the form $\neg arc_{i,i}$ for $1 \leq i \leq n$.

We have now forced all outgoing arcs to be active if the corresponding vertex is a successor. To guarantee that the instance is satisfiable if and only if there exists a tree decomposition of width at most k, we have to make sure by a cardinality constraint that at most k outgoing arcs of each vertex are active. A naive way of doing this is to add $\binom{n}{k+1}$ clauses of length $k+1$ for each vertex to ensure that there is no subset of $k+1$ active outgoing arcs. This, however, would result in $\mathcal{O}(n^{k+1})$ additional clauses. For this reason, we implement the cardinality constraint by a counter that is able to count the number of active outgoing arcs up to k.[1] Our counter is a slightly improved variant of the sequential unary counter proposed by Sinz [14]. Figure 3 illustrates two different counting scenarios with our encoding. In both cases the column on the left represents the outgoing arc variables associated to a vertex. The boxes on the right represent auxiliary variables of the counter, where each row represents a number up to k in unary encoding. In particular, the counter works in a bottom-up manner such that the number encoded in the i-th row is (at least) the number of active arcs up to row i. Every time we find an active arc, the counter is incremented as indicted by the arrows. If a variable in the last column is assigned true, the counter has reached its upper bound and we add clauses that forbid further active arcs as indicated by the shaded boxes in the scenario on the right. Note that we neglect the topmost row of counter variables as $n-1$ rows are sufficient. In summary, each counter requires $k(n-1) - k(k-1)/2$ additional variables and $(2k+1)n - k(k+3)$ clauses, which can both be estimated by $\mathcal{O}(kn)$. We denote the counter variables by $ctr_{i,j,l}$ with $1 \leq i \leq n$, $1 \leq j < n$, and $1 \leq l \leq \min(j, k)$, where i denotes the vertex the counter is associated to, j denotes the row, and l denotes the

[1] If $k > \lfloor n/2 \rfloor$, it is more efficient to count the number of inactive outgoing arcs up to $n - k$.

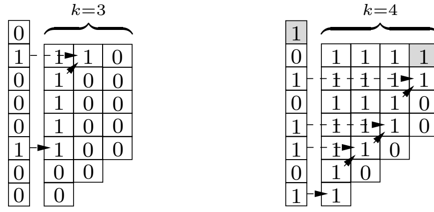

Fig. 3. Sequential unary counter that counts to 2 within its bound $k = 3$ (left) and that counts to 4 and exceeds its bound $k = 4$ as indicated by the shaded boxes (right)

column. The first kind of clauses added to encode the counter behavior are of the form $ctr_{i,j-1,l} \rightarrow ctr_{i,j,l}$ for $1 \leq i \leq n$, $1 < j < n$, and $1 \leq l \leq \min(j, k)$. These clauses ensure that the counter is never decremented, i.e., if some counter variable is assigned true, then all succeeding variables in the same column must be assigned true. Next we add clauses to enforce the actual counting. This is done by clauses of the form

$$arc_{i,j} \rightarrow ctr_{i,j,1}$$

for $1 \leq i \leq n$ and $1 \leq j < n$, which ensure that the counter is at least one if there is an active arc in the current row, and clauses of the form

$$(arc_{i,j} \wedge ctr_{i,j-1,l-1}) \rightarrow ctr_{i,j,l}$$

for $1 \leq i \leq n$, $1 < j < n$, and $1 < l \leq \min(j, k)$, which ensure that the counter is incremented from value $l - 1$ in the previous row to value l in the current row if there is an active arc in the current row. The effect of these clauses is indicated by the arrows in Figure 3. Finally, we enforce a conflict if any counter exceeds k by adding clauses of the form $\neg(arc_{i,j} \wedge ctr_{i,j-1,k})$ for $1 \leq i \leq n$ and $k < j < n$. Such a conflict is indicated by the shaded boxes in the right scenario of Figure 3.

Table 1 shows for several benchmark graphs from computational biology [7] the improvements of the bounds on the treewidth we have achieved with our encoding. To solve our SAT instances we used MiniSAT on a quad-core Intel Xeon CPU with 2.33GHz and 24GB RAM. The timeout for a single run was one hour. We observed that for graphs whose treewidth is known, the SAT solver could find a solution very fast if k was set to the treewidth and it took very long if k was set just below the treewidth. Thus, it can be seen as an indicator that there is no tree decomposition of width k if, for moderately sized graphs, the SAT solver does not find a solution within the timeout.

Table 1. Previous and new lower and upper bounds on the treewidth of four benchmark graphs. The last column shows the time to decide the obtained SAT instance with the new upper bound.

Graph	$\|V\|$	$\|E\|$	Prev LB [7]	Prev UB [7]	New LB	New UB	Time
1c75	69	683	28	30	28	29	15.9s
1dj7	73	743	26	27	26	26	134.0s
1dp7	76	769	25	27	25	26	118.3s
1en2	69	463	16	17	16	16	180.7s

4 Conclusion

Several theoretical and experimental attempts have been made in the past to attack the problem of determining the treewidth of graphs, but to the best of our knowledge this is the first time a SAT solver has been used for this. Although the graphs we used are relatively small and the improvements are not dramatic, we consider the presented approach as a first step towards computing the treewidth of graphs by SAT solvers. One of the main problems to be considered in future work is the size of the resulting SAT instances: Improved encodings that exploit the structure of a given graph may be able to keep the size of the instances small. Further work could be to identify redundant clauses that accelerate the SAT solving process and to adapt the encoding for related graph concepts like hypertree decomposition.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic and Discrete Methods 8(2), 277–284 (1987)
2. Bachoore, E.H., Bodlaender, H.L.: New upper bound heuristics for treewidth. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 216–227. Springer, Heidelberg (2005)
3. Bachoore, E.H., Bodlaender, H.L.: A branch and bound algorithm for exact, upper, and lower bounds on treewidth. Technical Report UU-CS-2006-012, Department of Information and Computing Sciences, Utrecht University (2006)
4. Bodlaender, H.L.: Discovering treewidth. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 1–16. Springer, Heidelberg (2005)
5. Bodlaender, H.L., Grigoriev, A., Arie, M.C., Koster, A.: Treewidth lower bounds with brambles. Algorithmica 51, 81–98 (2008)
6. Bodlaender, H.L., Koster, A.M.C.A., Wolle, T.: Contraction and treewidth lower bounds. Journal of Graph Algorithms and Applications (JGAA) 10(1), 5–49 (2006)
7. van den Broek, J.-W., Bodlaender, H.L.: TreewidthLIB (March 2009), http://people.cs.uu.nl/hansb/treewidthlib/
8. Clautiaux, F., Carlier, J., Moukrim, A., Nègre, S.: New lower and upper bounds for graph treewidth. In: Jansen, K., Margraf, M., Mastrolli, M., Rolim, J.D.P. (eds.) WEA 2003. LNCS, vol. 2647, pp. 70–80. Springer, Heidelberg (2003)
9. Courcelle, B.: Graph rewriting: An algebraic and logic approach. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B, ch. 5, pp. 193–242. Elsevier, Amsterdam (1990)
10. Dechter, R.: Tractable structures for constraint satisfaction problems. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 7, pp. 209–244. Elsevier, Amsterdam (2006)
11. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: Proc. of the 20th Conference on Uncertainty in Artificial Intelligence (UAI 2004). ACM International Conference Proceeding Series, vol. 70, pp. 201–208. AUAI Press (2004)
12. Koster, A.M.C.A., Bodlaender, H.L., van Hoesel, S.P.M.: Treewidth: Computational experiments. Electronic Notes in Discrete Mathematics 8, 54–57 (2001); Extended version available as Technical Report ZIB-Report 01-38, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
13. Robertson, N., Seymour, P.D.: Graph minors II. Algorithmic aspects of tree-width. Journal of Algorithms 7, 309–322 (1986)
14. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)