Available online at www.sciencedirect.com

SCIENCE DIRECT®

Discrete Optimization 3 (2006) 220–229

DISCRETE OPTIMIZATION

www.elsevier.com/locate/disopt

Tree decompositions of graphs: Saving memory in dynamic programming[☆,☆☆]

Nadja Betzler[a], Rolf Niedermeier[b,*], Johannes Uhlmann[a]

[a] *Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany*
[b] *Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany*

Received 1 October 2004; received in revised form 6 January 2005; accepted 1 May 2006
Available online 17 July 2006

Abstract

We propose a simple and effective heuristic to save memory in dynamic programming on tree decompositions when solving graph optimization problems. The introduced "anchor technique" is based on a tree-like set covering problem. We substantiate our findings by experimental results. Our strategy has negligible computational overhead concerning running time but achieves memory savings for nice tree decompositions and path decompositions between 60% and 98%.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Graph algorithms; (nice) Tree decompositions of graphs; Dynamic programming; Set covering; Memory usage

1. Introduction

Recently, tree decompositions of graphs have received considerable interest in practical applications and experiments [3,16–18,20,24,29,30]. Intuitively, due to its "tree-likeness" a graph with small treewidth often allows for efficient solutions for otherwise hard graph problems. The core tool in this solution process – besides constructing a tree decomposition of hopefully small width – is dynamic programming on tree decompositions [6,7,15]. As a rule, the running time of this dynamic programming is exponential with respect to the treewidth of the graph. Moreover, also the memory space consumption (in form of dynamic programming tables) is exponential with respect to the treewidth. Indeed, by our own practical experiences, the main bottleneck for the efficiency often is memory consumption and not running time. Koster et al. [17] point out that "Mainly due to limitations in computer memory, we are not able to solve the other instances".

☆ Supported by the Deutsche Forschungsgemeinschaft (DFG), research project "PEAL" (parameterized complexity and exact algorithms), NI 369/1, and Emmy Noether research group "PIAF" (fixed-parameter algorithms), NI 369/4. Main work done while all authors were with the Universität Tübingen.

☆☆ An extended abstract of this paper appeared in the proceedings of the Cologne Twente Workshop on Graphs and Combinatorial Optimization 2004, held in Loveno di Menaggio, Como, Italy, May 31–June 2, 2004.

* Corresponding author. Tel.: +49 3641 9 46320; fax: +49 3641 9 46002.
E-mail addresses: betzler@informatik.uni-tuebingen.de (N. Betzler), niedermr@minet.uni-jena.de (R. Niedermeier), johannes@informatik.uni-tuebingen.de (J. Uhlmann).

1572-5286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.disopt.2006.05.008

Attacking the "memory consumption problem" is the subject of recent research. Aspvall et al. [5] deal with the problem by trying to find an optimal traversal of the decomposition tree in order to minimize the number of dynamic programming tables stored simultaneously. Bodlaender and Fomin [9] theoretically investigate a new cost measure for tree decompositions and try to construct better tree decompositions in this way.

By way of contrast, in this empirically oriented paper we sketch a new approach that decreases the memory consumption by employing a (tree-like) set covering technique with some heuristic extensions. The point here is that our newly introduced "anchor technique" applies whenever, in principle, one needs to store *all* dynamic programming tables of a given tree decomposition. This is usually the case when one wants to solve the optimization version of a problem, that is, to construct actually an optimal solution or all of them. Here, the anchor technique tries to minimize the redundancy of information stored by avoiding to keep all dynamic programming tables in memory. Aspvall et al.'s [5] technique only seems to apply with respect to the decision version of a problem where one simply needs to do a bottom-up traversal of the decomposition tree and no second phase with a top-down traversal, as is typical in the case of optimization problems, is needed. Bodlaender and Fomin's [9] measure tries to minimize the cost when assuming that all tables need to be stored whereas we try to avoid storing all tables. Another possibility to prevent the storing of all tables is given by a naive "PSPACE-like" simulation: The first step consists of computing an optimal solution value for the root table and storing the corresponding pointers to the tables of its children. Next, one carries on recursively by repeating this process up to the level of the leaves. In this way, only all tables of *one* branch in the decomposition tree need to be stored at the same time. Obviously, this approach leads to a significantly increased running time whereas the increase of running time for our anchor technique is negligible. So far, according to our experiments the anchor technique seems to give the best results when dealing with path decompositions (with memory savings of around 95%) and nice tree decompositions (with memory savings of around 80%). We tested our technique when solving the NP-complete DOMINATING SET problem by dynamic programming on tree decompositions as described in [2,4]. The overhead concerning additional running time etc. for realizing the anchor technique is negligible.

2. Basic definitions and the anchor technique

2.1. Basic definitions

To describe our new technique, we first need to introduce some notation.

Definition 1. Let $G = (V, E)$ be an undirected graph. A *tree decomposition* of G is a pair $\langle\{X_i \mid i \in I\}, T\rangle$, where each X_i is a subset of V, called a *bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:

(1) $\bigcup_{i\in I} X_i = V$;
(2) for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$;
(3) for all $i, j, k \in I$, if j lies on the path between i and k in T, then $X_i \cap X_k \subseteq X_j$ (consistency property).

The *width* of $\langle\{X_i \mid i \in I\}, T\rangle$ equals $\max\{|X_i| \mid i \in I\} - 1$. The *treewidth* of G is the minimum k such that G has a tree decomposition of width k.

If the tree in Definition 1 is only a path, then we speak of a *path decomposition.* A tree decomposition with a particularly simple (and useful with respect to dynamic programming, cf., e.g., [15,2,4,14]) structure is given by the following.

Definition 2. A tree decomposition $\langle\{X_i \mid i \in I\}, T\rangle$ is called a *nice tree decomposition* if the following conditions are satisfied:

(1) Every node of the tree T has at most two children.
(2) If a node i has two children j and k, then $X_i = X_j = X_k$ (in this case i is called a JOIN NODE).
(3) If a node i has one child j, then either
    (a) $|X_i| = |X_j| + 1$ and $X_j \subset X_i$ (in this case i is called an INSERT NODE), or
    (b) $|X_i| = |X_j| - 1$ and $X_i \subset X_j$ (in this case i is called a FORGET NODE).

It is not hard to transform a given tree decomposition into a nice tree decomposition. More precisely, the following result holds (see [15, Lemma 13.1.3]).

Lemma 3. *Given a graph G and a tree decomposition of G that has width k and $O(n)$ nodes, where n is the number of vertices of G. Then, we can find a nice tree decomposition of G that has also width k and $O(n)$ nodes in time $O(n)$.*

We substantiate our findings using DOMINATING SET as concrete application problem for our experimental investigations. The NP-hard DOMINATING SET problem is defined as follows.

Input: An undirected graph $G = (V, E)$.
Task: Find a minimum size subset $V' \subseteq V$ such that every vertex from V is contained in V' or is adjacent to at least one vertex in V'.

The set V' is called a *dominating set*.

Tree decomposition based algorithms usually proceed in two phases. First, given some input graph, a tree decomposition of bounded width is constructed. Second, one solves the given problem (such as DOMINATING SET) using dynamic programming. Here, we are only concerned with the second step; see Bodlaender [8] for a recent survey concerning the first step. Dynamic programming to solve the optimization version of a problem again works in two phases—first bottom-up from the leaves to an arbitrarily chosen root node and then top-down from the root to the leaves in order to actually construct the solution.[1] To do this two-phase dynamic programming, however, in principle we need to store all the information gathered in this process. In particular, this means that one has to store all dynamic programming tables, each of them corresponding to a bag of the tree decomposition. Each bag B usually leads to a table that is exponential in its size. For instance, the table size in case of DOMINATING SET is $3^{|B|}$ since each vertex in a bag may be in one of three different *states* [26,2,4]. Clearly, even for modest bag sizes this leads to enormous memory consumption. This is the point where the anchor technique comes into play. It tries to avoid storing all dynamic programming tables by applying a set covering strategy.

The following two definitions are from [13].

Definition 4. Given a base set $S = \{s_1, s_2, \dots, s_n\}$ and a collection of subsets of S, $C = \{c_1, c_2, \dots, c_m\}$, $c_i \subseteq S$, we say that C is a *tree-like subset collection* if we can organize the subsets in C in a tree T such that every subset one-to-one corresponds to a node of T, and, for each $s_j \in S$, $1 \le j \le n$, all nodes in T corresponding to the subsets that contain s_j induce a subtree of T.[2]

Now, we can define a set covering problem where the inputs are restricted to tree-like subset collections, that is, TREE-LIKE WEIGHTED SET COVER (TWSC):

Input: A base set $S = \{s_1, s_2, \dots, s_n\}$ and a tree-like subset collection $C = \{c_1, c_2, \dots, c_m\}$ of S such that $\bigcup_{i=1}^{m} c_i = S$. Each subset in C has a real-valued weight $w(c_i) > 0$ for $1 \le i \le m$.
Task: Find a subset C' of C such that $\bigcup_{c \in C'} c = S$ and $\sum_{c \in C'} w(c)$ is minimum.

Note that the tree-likeness of a subset collection C of a base set S corresponds to the acyclicity of the hypergraph $H := (S, C)$ and, hence, it can be checked in linear time by means of the acyclicity test developed by Tarjan and Yannakakis [25].

2.2. The anchor technique

Trying to minimize space consumption, we have to obey that for *each* graph vertex we need to store important information. For instance, as to the DOMINATING SET, this information is either "being in the dominating set" or "being dominated but not being in the dominating set" or "not being dominated". Hence, on the one hand, we want to minimize the memory space consumed by the tables and, on the other hand, we have to make sure that no information is lost. In a natural way, this leads to a special WEIGHTED SET COVER problem: Each bag B of a tree

[1] For the decision version, the bottom-up phase suffices.
[2] This is nothing but the third condition of Definition 1 (consistency property).

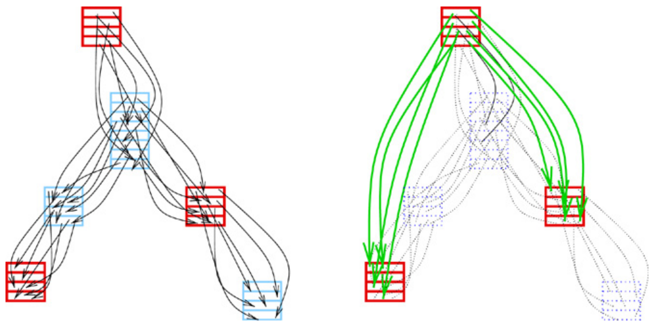

Fig. 1. Simplified tree structure with use of anchors. Herein, the anchors are drawn dark. The left-hand side shows a decomposition tree using all bags (tables), the right-hand side shows a decomposition tree only using the anchor bags (tables).

decomposition translates into a set containing its vertices and an associated weight exponentially depending on its table size (for DOMINATING SET, this is $3^{|B|}$). Losing no information then simply means that we have to cover the base set consisting of all graph vertices V by a selection of the "bag sets". To use as little memory as possible then means to do the selection of the bag sets such that their total weight is minimized while keeping each vertex from V covered. This is nothing but a TREE-LIKE WEIGHTED SET COVER (TWSC) problem with exponential weight distribution. This formalization inspired theoretical work on TWSC [13] with applications also in computational biology [23]. There is a polynomial-time algorithm in case of paths (instead of trees) and a fixed-parameter tractability result for the general version. More precisely, the following results are known [13]:

- TWSC is polynomial-time solvable with respect to unit weights ($O(m \cdot n)$ running time).
- TWSC is NP-hard in the general case and also with respect to exponential weight distribution.
- If the underlying tree is restricted to be a path (in this case we refer to it as PATH-LIKE WEIGHTED SET COVER), the problem has been studied under the name SET COVER WITH CONSECUTIVE ONES PROPERTY (SCC1P) and can be solved in polynomial time ($O(m^2 \cdot n)$) [27].
- TWSC is fixed-parameter tractable with respect to parameter $k :=$ "maximum subset size", that is, TWSC can be solved in $O(3^k \cdot m \cdot n \cdot l)$ time, where $l < m$ is the maximum degree of a node in the tree of the underlying tree-like subset collection.

Note that the anchor technique as described here can be applied for every weight function as defined for TWSC. In particular, it also applies to combinatorial problems in which the number of states depends on the vertex—for an example, consider the CONNECTED DOMINATING SET problem where one asks for a dominating set which forms a connected component.

As it turns out, in our concrete application scenario already simple polynomial-time computable data reduction rules suffice to obtain optimal solutions of TWSC in a fast way (see Section 3.1 for more details).

We call the bags that are in the above described set cover solution C' *anchors*. After finding a set of anchors, we can simplify the decomposition tree with respect to the memory requirements by setting our pointers directly from anchor to anchor. This means that, apart from the table involved in the current update step between neighboring bags, we do not have to save any bag tables that are not anchors. In this context, however, it is important to observe that each bag table row has an entry pointing to all of the bag's child bags. Fig. 1 gives a graphical illustration of the pointer reduction by using anchors.

More precisely, considering the pointers representing the (simplified) tree structure between the rows of the dynamic programming tables, we have to differentiate two cases: First, if the bag of a child of a tree node is an anchor, then we set our pointer in the normal way. If the bag of a child is not an anchor, then we use the pointer from the corresponding row of the child's bag table: Assume that the value for row c in the table of a bag was computed using row c' in a child table which is not an anchor and that row c' has an entry pointing to c'', then instead of setting the pointer of c to c' (as is usual in the case without the use of anchors) we set the pointer to c''. Finally, the table containing c' can be removed. Note that, assuming that we perform this restructuring process in a bottom-up manner, the table that contains c'' is always an anchor. If c' points to more than one anchor, we have to set pointers to all of them. Observe that the bag table of the root is treated in a special way. We do not have to store it because we are only interested in *one* entry with optimal value. So, we can assume that all vertices of the root bag are already covered. This concludes the description of the basic ideas of the *anchor technique*.

At this point, we have to emphasize one important fact concerning the applicability of the anchor technique as described here. First, note that with respect to path decompositions the goal to minimize the overall memory consumption has its one-to-one correspondence in the optimization goal of the PATH-LIKE WEIGHTED SET COVER problem as described above. Going to trees, however, the formalization as TREE-LIKE WEIGHTED SET COVER does not take into account the additional costs that are due to the modified pointer structure caused by the anchor technique. Hence, to somewhat mitigate this effect, in our experiments, which we report on in Section 4, we further used a heuristic to modestly increase the set of anchors with the goal to decrease the overhead caused by the pointers. This will be further discussed in Section 3.2.

3. Algorithmic aspects of the anchor technique

In order to compute the set of anchors we first find – by the use of simple data reduction rules – a solution for the TWSC instance where each bag is associated with a weight which has a value exponential in its table size, and then we apply a heuristic to find so-called *help anchors* if necessary. These help anchors shall mitigate the effects caused by the pointer structure modifications as discussed in the last paragraph of Section 2.

3.1. Data reduction rules for TWSC

Let $\langle \{X_i \mid i \in I\}, T \rangle$ be a tree decomposition of graph $G = (V, E)$ with $T = (I, F)$ and a weight function $w : \{X_i \mid i \in I\} \to \mathbb{R}^+$. We consider a TWSC instance with base set V and tree-like subset collection $\{X_i \mid i \in I\}$. We implemented the following simple and efficient data reduction rules for TWSC. The data reduction process works as long as possible, that is, until no reduction applies any more. If possible the rules determine bags from $\{X_i \mid i \in I\}$ that must be part of an optimal TWSC solution and return a reduced TWSC instance consisting of bags that contain uncovered vertices (if there are any). Note that the weight initially assigned to a bag is not changed by any reduction rule.

Rule 1: If a bag X_i contains at least one element that is not contained in any other bag, then put X_i into the solution and remove all elements that occur in X_i from all bags of the tree decomposition.

Rule 2: If there is an edge $e = \{i, j\} \in F$ such that

(1) $X_i \subseteq X_j$ and
(2) $w(X_i) \geq w(X_j)$

then connect each neighboring node of i (except for j) with j and delete i together with its incident edges from T.

Rule 3: If for any bag X_i the fact $a \in X_i$ implies $b \in X_i$, then remove b from the bags of the tree decomposition.

Rule 4: If there is an edge $e = \{i, j\} \in F$ such that $X_i \cap X_j = \emptyset$, then remove this edge from F.

Rule 5: If for a bag X_i there is a set of nodes $j_1, j_2, \ldots, j_l \in I$, such that

(1) $X_i \subseteq \bigcup_{k=1}^{l} X_{j_k}$,
(2) $w(X_i) \geq \sum_{k=1}^{l} w(X_{j_k})$, and
(3) the neighbors $N(i)$ of i form a tree-like subset collection,

then remove i from I and connect its neighbors such that the tree-likeness is not violated.

Note that Rule 2 and Rule 4 may apply to an instance which already has been reduced by application of other reduction rules. The correctness of the above rules is easy to see. Rule 5 has only been implemented restricted to the special case that node i has *two* neighbors which cover X_i. In this case, the neighbors can always be connected without violating the consistency property of tree decompositions (since in any case a set of two bags is tree-like). Because of the NP-completeness of TWSC in general the above polynomial-time data reduction rules do not suffice in order to obtain an optimal solution for TWSC for all possible inputs. Nevertheless, in many cases of our experiments the instances were completely solved in this way, and, if not so, the remaining “reduced” instances were small enough (or split into small enough independent pieces, respectively) such that a simple exhaustive search algorithm could efficiently be applied.[3] We mention in passing that similar observations concerning the effectiveness of simple polynomial-time data reduction rules in the context of domination-like or set covering problems were made in [1,21,28].

3.2. Help anchors

In this section we present a heuristic to determine a set of "help anchors" to mitigate the effects caused by neglecting the pointer structures in the TWSC formalization. Let $\langle X = \{X_i \mid i \in I\}, T = (I, F)\rangle$ be a tree decomposition and $A \subseteq I$ be a set of anchors, that is, a selection of the bags such that all vertices are covered ($\bigcup_{i \in A} X_i = V$). So, we call the set

$$C_i := \{j \in I \mid j \text{ is the first anchor on a path from } i \text{ to a leaf}\}$$

the *anchor children* of node i. Furthermore, if we refer to the number of children of node i as outdeg(i), let

$$\text{max-table-size} := \max\{w(X_i) \cdot \text{outdeg}(i) \mid i \in I\}$$

be the maximum space consumption of a table without the use of anchors.

We further comment on the problems – as already stated in the last paragraph of Section 2 – with the TWSC formalization in the context of finding suitable anchors. There are two main problems in case of trees instead of paths which are put down to the memory costs for the pointers (reflecting the (modified) tree structure) and which are neglected in the TWSC formalization.

The first problem concerns the overall memory consumption in the end of the bottom-up phase of dynamic programming (that is the information that is needed to trace back a solution). In fact, formulating the problem in terms of the Dominating Set problem, to get the best savings concerning this memory consumption one should minimize the sum

$$\sum_{i \in A} 3^{|X_i|} \cdot |C_i|$$

where A denotes the set of anchors (whose union of the corresponding bags still has to cover the set of all graph vertices) and C_i denotes the set of *anchor children* of node i in the tree decomposition. More generally, the goal is to minimize $\sum_{i \in A} w(X_i) \cdot |C_i|$ where $w(X_i)$ basically reflects the size of table X_i. The multiplicative factor $|C_i|$ is ignored in the formalization as TWSC.

The other problem with the TWSC formalization arises from the fact that we have to store all pointers of every (not only anchors) bag table during one step in the local update process. It turned out to be a crucial source of memory demands: In some tree decompositions we find a set of anchor bags that consists only of the bags of the leaves and the root. In this case the children of the root (which usually have big bag tables) have to store pointers to all leaves in the tree decomposition during the local update process.

To mitigate the effect of the additional factor $|C_i|$ and to ensure that we do not even increase the total amount of memory requirement, we designed a heuristic that extends an anchor set solution found by the TWSC optimization. In a few words, the central objective of our heuristic is to avoid to store – even temporarily – a table (including its pointers) which has a memory consumption greater than the maximum space consumption of any table of the tree decomposition without anchors.

More specifically, the idea for finding the help anchors is as follows. Consider a node i in the decomposition tree that has more anchor children than "normal" children, i.e., $|C_i| \geq \text{outdeg}(i)$. If the table of i has a space consumption greater than the maximum space consumption of a table without anchors (that is, $w(X_i) \cdot |C_i| > \text{max-table-size}$), we make use of additional bags (called *help anchors*) from the descendants of i in order to decrease the number of anchor children of X_i. Furthermore, the heuristic additionally chooses subsets of vertices as help anchors that have more (help) anchor children than original children. It tries several possibilities for this choice and simply picks out the one with lowest memory requirement. An interesting observation is that we sometimes can decrease the space consumption by adding more help anchors.

[3] Alternatively, the fixed-parameter algorithm from [13] or other sophisticated approaches such as integer linear programming could be applied. In the experiments which are presented subsequently, this was not necessary.

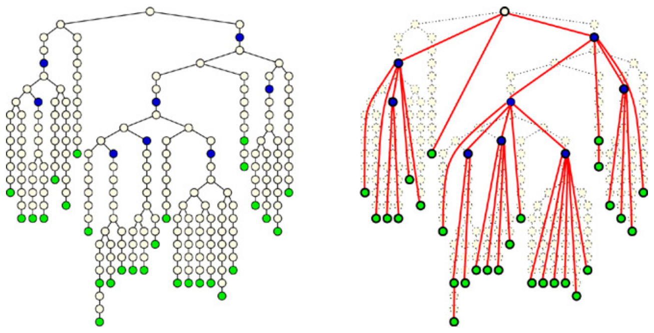

Fig. 2. Screen-shot of the original decomposition tree (left-hand side) and its simplification by the anchor technique (right-hand side). The grey shaded nodes are anchors and the dark shaded nodes are help anchors. Instead of storing the whole structure (left-hand side) one only has to store the tables of the anchors and help anchors and the pointers from (help) anchor to (help) anchor.

It is conceivable, however, that this heuristic for finding help anchors will not be the final word to say in this respect. Future (experimental) work has to show whether better strategies are available. As also indicated by the experiments presented in Section 4, the anchor technique seems to be least effective in case of tree decompositions with some big bags close to the root and lots of small anchor leaves. These difficulties also seem to be the reason that the technique so far achieves no satisfactory improvements in memory consumption when dealing with "normal" (that is, non-nice) tree decompositions. On the positive side, however, with anchors one can "always" afford to use nice tree decompositions instead of normal ones without loss of (memory) efficiency. Observe that nice tree decompositions significantly simplify the dynamic programming as concretely exhibited in the cases of Dominating Set [2,4], Connected Dominating Set [11], and Power Dominating Set [14].

4. Computational analysis and results

We performed empirical tests with the anchor technique on path and nice tree decompositions. If not stated otherwise, our tests were run on a standard Linux PC with one gigabyte main memory and a 2.26 GHz processor using the LEDA library [22]. We implemented the data reduction rules from Section 3.1 (combined with, if necessary, fast exhaustive search) to find anchors, that is, to solve the corresponding TWSC problem. Having the anchors at hand, it then is rather simple to adapt the pointer structure between the dynamic programming tables as described before. Without going into details, let us only mention that the overhead for adding the pointer technique turned out to be negligible in terms of the overall running time of dynamic programming—we found this when doing tests with Dominating Set. Thus, to start with a summary of our empirical findings, we may say that the anchor technique

- is easy to implement,
- costs very little additional running time, and
- leads to significant savings concerning memory usage in case of path decompositions and nice tree decompositions.

Fig. 2 shows a screen-shot of a tree decomposition together with anchors as produced by our software system realizing the anchor technique together with tree decomposition based algorithms for hard graph problems.

To generate tree decompositions for our test graph instances, we basically used the same heuristics as described in [10,16]. We briefly mention in passing that the running times consumed by the anchor technique never exceeded few seconds, whereas running dynamic programming for Dominating Set took time in the range of up to minutes for graphs with treewidth smaller than 14.

We begin with first results on path decompositions of grid graphs.[4] A grid graph simply is a graph of maximum vertex degree four whose vertices and edges form an $r \times s$ grid for positive integers r and s in the canonical way. Recall that for path decompositions we have argued in the last paragraph of Section 2 that the formalization as Path-Like Weighted Set Cover is perfectly alright. Thus, no help anchors are needed. Not surprisingly, hence, in this case we obtained the strongest memory saving of more than 90%. Table 1 shows our results in more detail.

[4] For grid graphs an optimal path decomposition is an optimal tree decomposition as well.

Table 1
Memory savings for 10 × 10, 10 × 30, and 10 × 50 grids, assuming a table size of $3^{|B|}$ for each bag B (as also shown in the subsequent tables)

Grid graphs				Path decomposition			Nice path decomposition		
Name	$\|V\|$	$\|E\|$	pw	nob	noa	mem (%)	nob	noa	mem (%)
`grid_10_10`	100	180	10	89	9	90.9	178	9	97.7
`grid_10_30`	300	560	10	289	27	91.0	578	29	97.6
`grid_10_50`	500	940	10	489	45	91.0	978	49	97.5

Here, pw denotes the width of the underlying path decomposition, nob denotes the number of bags, noa denotes the number of anchors, and mem denotes the percentage of memory saved.

Table 2
Results for graphs generated with the BRITE tool

$\|V\|$	$\|E\|$	nob	noa	maxb	maxa	nomb	av10b	al	noh	maxh	avh	mem (%)
100	197	579.0	55.5	12.3	4.5	24.1	11.5	52.1	12.8	12.1	10.1	77.2
150	297	1023.9	84.7	16.8	4.5	25.9	15.2	80.3	20.8	16.7	12.0	74.0
200	397	1573.7	114	21.1	4.4	24.6	18.7	108.1	25.8	20.6	14.4	79.9

Each row represents the average numbers taken over 15 graphs each time. We only consider nice tree decompositions here. Besides the figures used in Table 1, we additionally measured maxb (maximum bag size; equivalent to treewidth +1), maxa (maximum anchor size), nomb (number of maximum size bags), av10b (average size of the 10 percent biggest bags), al (number of anchors which are leaves), noh (number of help anchors), maxh (size of maximum help anchor), and avh (average size of help anchors).

Table 3
Results for combinatorially random graphs generated with LEDA (single graphs, no average numbers)

$\|V\|$	$\|E\|$	nob	noa	maxb	maxa	nomb	av10b	al	noh	maxh	avh	mem (%)
200	415	1595	84	44	6	6	36.2	74	13	42	28.0	93.1
200	371	1459	88	38	6	8	32.1	73	16	38	28.5	65.3
200	425	1662	91	44	6	7	36.9	83	23	44	22.1	62.5
200	372	1484	95	37	6	4	31.1	79	21	36	18.8	79.1
200	410	1636	82	49	6	6	40.3	73	13	48	33.9	83.1
200	415	1735	83	43	6	14	38.4	76	12	43	32.3	76.3

Again only nice tree decompositions are considered. Same figures as in Table 2.

As to nice tree decompositions, we performed tests with so-called "Internet graphs" as produced by the BRITE topology generator [19] and combinatorially random graphs generated by LEDA [22]—both yielding sparse graphs. Here, to achieve a significant progress in comparison with ordinary dynamic programming, we had to extend the anchor technique by the help anchor heuristic as described in Section 3.2.[5] With this heuristic, some more bags are added to the set of anchors in order to deal with otherwise increased memory requirements due to the pointer structure. Doing so, we achieved memory savings of around 80%, see Tables 2 and 3.

The following experiments were run on a standard Linux PC with a 2.4 GHz processor, 512 KB cache, and 1 GB main memory.

Furthermore, we considered nice tree decompositions of real-world graphs from Bodlaender's TreewidthLIB library[6] which provides a benchmark for tree decomposition based algorithms. We investigated 137 graphs and achieved even better results than for the synthetically generated graphs. On average we could achieve memory savings of 86.6%. In Table 4 we show some results for graphs with tree decompositions with a size reasonable for dynamic programming for DOMINATING SET. Furthermore, it seems to be interesting that there are very few instances for which the anchor technique seems not to allow for a great improvement. Out of all 137 graphs there was only one instance with memory savings of less than 50%—whereas for 49 instances we could save more than 99% of memory.

[5] In fact, without the additional help anchor heuristic the memory usage was even worse in some cases.

[6] See http://www.cs.uu.nl/people/hansb/treewidthlib/index.php.

Table 4
Results for graphs obtained from Bodlaender's TreewidthLIB

Name	$\|V\|$	$\|E\|$	nob	noa	maxb	maxa	nomb	av10b	al	noh	maxh	avh	mem (%)
163	80	508	288	39	11	10	9	9.81	24	3	10	7.67	86.2
219	189	366	660	83	12	6	7	9.98	56	13	12	8.62	60.3
259	74	211	268	34	13	7	5	11.6	21	5	12	9.8	80.6
315	23	71	97	10	11	6	20	11	10	2	11	10.5	90.1
402	41	195	78	7	12	9	6	11.8	3	0	0	0	99.6
567	76	218	161	17	12	11	62	12	4	0	0	0	95.1

Again only nice tree decompositions are considered. Same figures as in Table 2; additionally, the name of the graphs from the TreewidthLIB is given.

At last, we like to mention that the anchor technique allowed us to solve instances that have not been solved by our dynamic programming before. For example, applying the anchor technique we could compute an optimal dominating set of a grid graph of size 12×50 whereas before we could not even compute a solution for a 12×12 grid graph.

5. Conclusion

We introduced the anchor technique to significantly reduce the space consumption of dynamic programming on (nice) tree decompositions of graphs for solving (hard) optimization problems. We only mention in passing that our technique not only applies to path/tree decomposition based dynamic programming but also appears to be useful for dynamic programming on branch decompositions (see [12] for a recent algorithm employing branch decompositions). The computational and implementation overhead caused by our method was negligible in all our experiments.

In future work, one might try to further mitigate the gap between the formalization of the anchor idea in terms of TREE-LIKE WEIGHTED SET COVER and the practically relevant point of keeping the tree/table pointer structure reasonably simple—recall that the pointer structure is neglected by the TREE-LIKE WEIGHTED SET COVER formalization. Finally, it would be interesting to see how our technique combines with the saving techniques from [5,9].

Acknowledgments

We thank Jochen Alber, Frederic Dorn, Jiong Guo, and Falk Hüffner for helping with theoretical as well as practical considerations. We are grateful to two anonymous referees for helpful comments that improved the presentation.

References

[1] J. Alber, N. Betzler, R. Niedermeier, Experiments on data reduction for optimal domination in networks, in: Proceedings 1st INOC, 2003, pp. 1–6 (Long version to appear in Annals of Operations Research).
[2] J. Alber, H.L. Bodlaender, H. Fernau, T. Kloks, R. Niedermeier, Fixed parameter algorithms for Dominating Set and related problems on planar graphs, Algorithmica 33 (4) (2002) 461–493.
[3] J. Alber, F. Dorn, R. Niedermeier, Experimental evaluation of a tree decomposition based algorithm for Vertex Cover on planar graphs, Discrete Applied Mathematics 145 (2) (2005) 219–231.
[4] J. Alber, R. Niedermeier, Improved tree decomposition based algorithms for domination-like problems, in: Proceedings 5th LATIN 2002, in: LNCS, vol. 2286, Springer-Verlag, 2002, pp. 613–627.
[5] B. Aspvall, A. Proskurowski, J.A. Telle, Memory requirements for table computations in partial k-tree algorithms, Algorithmica 27 (3) (2000) 382–394.
[6] H.L. Bodlaender, Treewidth: Algorithmic techniques and results, in: Proceedings 22nd MFCS, in: LNCS, vol. 1295, Springer-Verlag, 1997, pp. 19–36.
[7] H.L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theoretical Computer Science 209 (1998) 1–45.
[8] H.L. Bodlaender, Discovering treewidth, in: Proceedings 31st SOFSEM, in: LNCS, vol. 3381, Springer-Verlag, 2005, pp. 1–16.
[9] H.L. Bodlaender, F.V. Fomin, Tree decompositions with small cost, Discrete Applied Mathematics 145 (2) (2005) 143–154.
[10] H.L. Bodlaender, A.M.C.A. Koster, F. van den Eijkhof, Pre-processing rules for triangulation of probabilistic networks, Technical Report, Institute of Information and Computing Science, Utrecht University, 2003.
[11] E.D. Demaine, M. Taghi Hajiaghayi, Bidimensionality: new connections between FPT algorithms and PTASs, in: Proceedings 16th ACM-SIAM Symposium on Discrete Algorithms, SODA, 2005, pp. 590–601.
[12] F.V. Fomin, D.M. Thilikos, Dominating sets in planar graphs: branch-width and exponential speed-up, in: Proceedings 14th ACM-SIAM Symposium on Discrete Algorithms, SODA, 2003, pp. 168–177.

[13] J. Guo, R. Niedermeier, Exact algorithms and applications for Tree-like Weighted Set Cover, Journal of Discrete Algorithms (2005) (in press).
[14] J. Guo, R. Niedermeier, D. Raible, Improved algorithms and complexity results for power domination in graphs, in: Proceedings 15th FCT, in: LNCS, vol. 3623, Springer-Verlag, 2005, pp. 172–184.
[15] T. Kloks, Treewidth. Computations and Approximations, in: LNCS, vol. 842, Springer-Verlag, 1994.
[16] A.M.C.A. Koster, H.L. Bodlaender, S.P.M. Hoesel, Treewidth: Computational Experiments, in: Electronic Notes in Discrete Mathematics, vol. 8, Elsevier Science Publishers, 2001.
[17] A.M.C.A. Koster, S.P.M. van Hoesel, A.W.J. Kolen, Solving Frequency Assignment Problems via Tree-decomposition, in: Electronic Notes in Discrete Mathematics, vol. 3, Elsevier Science Publishers, 1999.
[18] A.M.C.A. Koster, S.P.M. van Hoesel, A.W.J. Kolen, Solving partial constraint satisfaction problems with tree decompositions, Networks 40 (3) (2002) 170–180.
[19] A. Medina, A. Lakhina, I. Matta, J. Byers, BRITE: An approach to universal topology generation, in: Proceedings MASCOTS 2001, IEEE Computer Society, 2001, August.
[20] J.-F. Manouvrier, C. Lucet, Resolving the network reliability problem with a tree decomposition of the graph, in: Proceedings 1st OPODIS, Hermes, 1997, pp. 193–204.
[21] S. Mecke, D. Wagner, Solving geometric covering problems by data reduction, in: Proceedings 12th ESA, in: LNCS, vol. 3221, Springer-Verlag, 2004, pp. 760–771.
[22] K. Mehlhorn, S. Näher, LEDA: A Platform of Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, England, 1999.
[23] R.D.M. Page, J.A. Cotton, Vertebrate phylogenomics: reconciled trees and gene duplications, in: Proceedings Pacific Symposium on Biocomputing 2002, 2002, pp. 536–547.
[24] Y. Song, C. Liu, R. Malmberg, F. Pan, L. Cai, Tree decomposition based fast search of RNA structures including pseudoknots in genomes, in: Proceedings IEEE Computational Systems Bioinformatics Conference, 2005.
[25] R.E. Tarjan, M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM Journal on Computing 13 (3) (1984) 566–579.
[26] J.A. Telle, A. Proskurowski, Practical algorithms on partial k-trees with an application to domination-like problems, in: Proceedings 3rd WADS'93, in: LNCS, vol. 709, Springer-Verlag, 1993, pp. 610–621.
[27] A.F. Veinott, H.M. Wagner, Optimal capacity scheduling, Operations Research 10 (1962) 518–532.
[28] K. Weihe, Covering trains by stations or the power of data reduction, in: Proceedings 1'st ALEX '98, 1998, pp. 1–8.
[29] T. Wolle, A framework for network reliability problems on graphs of bounded treewidth, in: Proceedings 13th ISAAC 2002, in: LNCS, vol. 2518, Springer-Verlag, 2002, pp. 137–149.
[30] J. Xu, F. Jiao, B. Berger, A tree-decomposition approach to protein structure prediction, in: Proceedings IEEE Computational Systems Bioinformatics Conference, 2005.