Tree Decompositions of Graphs: Saving Memory in Dynamic Programming [1]

Nadja Betzler Rolf Niedermeier Johannes Uhlmann

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany
E-mail: {betzler,niedermr,johannes}@informatik.uni-tuebingen.de

Abstract

We propose an effective heuristic to save memory in dynamic programming on tree decompositions when solving graph optimization problems. The introduced "anchor technique" is closely related to a tree-like set covering problem.

1 Introduction

Recently, tree decompositions of graphs have received considerable interest in practical applications. Intuitively, a graph with small treewidth allows for efficient solutions of 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 [3]. As a rule, both the running time of this dynamic programming and the memory space consumption are exponential with respect to the treewidth. Indeed, also by our own practical experiences, the main bottleneck often is memory consumption rather than running time.

Attacking the "memory consumption problem" is subject of recent research. For instance, Aspvall et al. [2] 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 [4] 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 experimentally oriented paper we sketch a new approach that tries to decrease the memory consumption by employing a (tree-like) set covering technique with some heuristic extensions. The point here is that our so-called "anchor technique" applies whenever, in principle, one needs to store *all* dynamic programming tables of a given tree decomposition. This is usually

[1] Supported by the Deutsche Forschungsgemeinschaft (DFG), research project "PEAL," NI 369/1, and junior research group "PIAF," NI 369/4.

the case when one wants to solve the optimization version of problems, i.e., to actually construct 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.[2] 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%).

2 Basic Ideas and the Anchor Technique

To describe our new technique, we first need to introduce some notation [3]. Let $G = (V, E)$ be a 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: $\bigcup_{i \in I} X_i = V$; for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$; and for all $i, j, k \in I$, if j lies on the path between i and k in T, then $X_i \cap X_k \subseteq X_j$. 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 above tree 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. [1,3]) structure is given by the following. A tree decomposition is called a *nice tree decomposition* if the following conditions are satisfied: Every node of the tree T has at most two children; if a node i has two children j and k, then $X_i = X_j = X_k$; and if a node i has one child j, then either $|X_i| = |X_j| + 1$ and $X_j \subset X_i$ or $|X_i| = |X_j| - 1$ and $X_i \subset X_j$. It is not hard to transform a given tree decomposition into a nice tree decomposition.

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. Dynamic programming to solve the optimization version of a problem again works in two phases—first bottom-up from the leaves to the root (which can be chosen arbitrarily) and then top-down from the root to the leaves in order to actually construct the solution. To do this two-phase dynamic programming, however, 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

[2] Aspvall et al.'s [2] technique only seems to apply with respect to the decision version of a problem where one only needs to do a bottom-up traversal of the decomposition tree. Bodlaender and Fomin's [4] measure tries to minimize the cost when assuming that all tables need to be stored whereas we try to avoid storing all tables.

Dominating Set is $3^{|B|}$ [1]. Clearly, even for modest bag sizes this leads to enormous memory consumption. This is the point where the anchor technique comes into play.

Trying to minimize space consumption, we have to obey that for *each* graph vertex we need to store important information. 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 translates into a set containing its vertices and an associated weight exponentially depending on its table size (for Dominating Set, this is $3^{|B|}$). Loosing 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 with applications also in computational biology [5]. In our setting already simple data reduction rules suffice to obtain optimal solutions of TWSC in a fast way. We call the bags in the set cover *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. That means, apart from the table involved in the current update process, we do not have to store any bag tables that are not anchors.

Note that with respect to path decompositions the goal of minimizing the overall memory consumption has a one-to-one correspondence to the optimization goal of the Path-like Weighted Set Cover problem as described above. Going to trees, however, the formalization as TWSC does not take into account the additional costs that are due to the modified pointer structure caused by the anchor technique (cf. Section 3).

3 Computational Analysis and Results

We performed some first empirical tests of the anchor technique on path and nice tree decompositions. To do so, we implemented some simple and efficient polynomial-time data reduction rules to find anchors. Having the anchors at hand, it then is rather simple to adapt the pointer structure between the dynamic programming tables. Without going into details, let us only mention that the algorithmic and implementation overhead for adding the pointer technique turned out to be negligible in terms of the overall running time of dynamic programming.[3] Thus, to start with a summary of our empirical findings, we may say that the anchor technique is easy to implement, it costs

[3] We found this when doing tests with Dominating Set.

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	89.88%	179	17	93.26%
grid_10_30	300	560	10	289	27	90.66%	578	44	94.81%
grid_10_50	500	940	10	489	45	90.80%	978	58	95.98%

Table 1

We compare the memory savings for three different grid graphs, assuming a table size of $2^{|B|}$ for each bag B, as we also do in the subsequent tables. (Assuming size $3^{|B|}$ would even improve the saving effect.) Herein, 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.

$\|V\|$	$\|E\|$	nob	noa	maxb	maxa	nomb	av10b	al	noh	maxh	avh	mem
100	197	578.8	55.5	12.3	4.5	24.1	11.5	52.1	13.8	12.1	9.9	74.9 %
150	297	1023.9	85.1	16.8	4.9	25.9	15.2	80.3	21.5	16.7	12.0	73.5 %
200	397	1573.7	114	21.1	4.5	24.6	18.7	108.1	28.1	20.9	13.8	77.8%

Table 2

We compare graphs generated with the BRITE tool. 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), 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, i.e., additional anchors found by the heuristic), maxh (size of maximum help anchor), and avh (average size of help anchors).

$\|V\|$	$\|E\|$	nob	noa	maxv	maxa	nomv	av10v	al	noh	maxh	avh	mem
200	415	1595	84	44	6	6	36.2	74	13	42	28	87.2%
200	371	1459	88	38	6	8	32.1	73	16	38	28.5	63.4 %
200	425	1662	93	44	7	7	36.9	83	23	44	22.1	66.4%
200	372	1484	95	37	6	4	31.1	79	22	36	19.6	74.9%
200	410	1636	82	49	6	6	40.3	73	13	47	33.8	87.4%
200	415	1735	83	43	6	14	38.4	76	21	43	26.5	75.1%

Table 3

We compare random graphs generated with LEDA (single graphs, no average numbers). Again only nice tree decompositions are considered. Same figures as in Table 2.

very little additional running time, and it leads to significant savings concerning memory use. We begin with first results on path decompositions of grid graphs.[4] In this case, we obtained the strongest memory saving of around 90–95% and more. Table 1 shows our results in more detail.

The problem with TWSC in case of trees instead of paths lies in the memory costs for the pointers (reflecting the (modified) tree structure) which are neglected in the TWSC formalization. In fact, formulating the problem in terms of the Dominating Set problem, to get the best savings concerning memory one should minimize the sum $\sum_{B\in A} 3^{|B|} \cdot |C|$, where A denotes the

[4] Here an optimal path decomposition is an optimal tree decomposition as well.

set of anchors and C denotes the set of "anchor children" of B in the tree decomposition. The second multiplicative factor is ignored in the formalization as TWSC and it turned out to be a crucial source of memory demands. To mitigate this effect, we designed a heuristic that extends an anchor set solution found by the TWSC optimization. In few words, the central objective of our heuristic is that we do not have to store a table (including its pointers) which has a memory consumption greater than the maximum memory consumption of a bag of the tree decomposition without anchors. We defer the details to the full paper. As to (nice) tree decompositions, we performed tests with so-called "Internet graphs" as produced by the BRITE topology generator and random graphs generated by the LEDA library. We achieved memory savings of around 80%, see Tables 2 and 3. The worst cases for the anchor technique seem to be 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 not always achieves satisfactory improvements on memory consumption when dealing with "normal" (i.e., 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. Nice tree decompositions significantly simplify the dynamic programming as exhibited in the case of Dominating Set [1].

To summarize, the anchor technique significantly reduces the space consumption of dynamic programming on tree decompositions of graphs for solving (hard) optimization problems. Our technique not only applies to path/tree decomposition based dynamic programming but also appears to be useful for dynamic programming on branch decompositions. The time overhead caused by our method was negligible in all our experiments. In ongoing and future work, we try to extend our experiments to further classes of graphs. We will also 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.

References

[1] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, **33(4)**: 461–493, 2002.

[2] B. Aspvall, A. Proskurowski, and J. A. Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica* **27**: 382–394, 2000.

[3] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proceedings 22nd MFCS*, Springer-Verlag LNCS 1295, pp. 19–36, 1997.

[4] H. L. Bodlaender and F. V. Fomin. Tree decompositions with small cost. In *Proceedings 8th SWAT*, Springer-Verlag LNCS 2368, pp. 378–387, 2002.

[5] J. Guo and R. Niedermeier. Exact algorithms for Tree-like Weighted Set Cover. Manuscript, submitted for publication, February 2004.