Space-efficient construction variants of dynamic programming

Hans L. Bodlaender

Jan Arne Telle

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-030

www.cs.uu.nl

Space-efficient construction variants of dynamic programming

Hans L. Bodlaender* Jan Arne Telle†

Abstract

Many dynamic programming algorithms that solve a decision problem can be modified to one that solves the construction variant of the problem by additional bookkeeping and going backwards through stored answers to subproblems. It is also well known that for many dynamic programming algorithms, one can save memory space by throwing away tables of information that is no longer needed, thus reusing the memory. Somewhat surprisingly, the observation that these two modifications cannot be combined is frequently not made. In this paper we consider the case of dynamic programming algorithms on graphs of bounded treewidth. We give algorithms to solve the construction variants of such problems that use only twice the amount of memory space of the decision versions, with only a logarithmic factor increase in running time. Using the concept of strong directed treewidth we then discuss how these algorithms can be applied to dynamic programming in general.

Keywords: Algorithms and data structures; dynamic programming; memory use of algorithms; treewidth.

1 Introduction

Dynamic Programming (DP) is one of the most common algorithmic techniques. It is well known that many dynamic programming algorithms that solve a decision problem can be modified to one that solves the construction variant of the problem by additional bookkeeping and going backwards through stored answers to subproblems. It is also well known that for many dynamic programming algorithms, one can save memory space by throwing away tables of information that is no longer needed, thus reusing the memory. Somewhat surprisingly, the observation that these two modifications cannot be combined is frequently not made: when the 'text-book' modification to save memory is made to a dynamic programming algorithm, the information to construct solutions is deleted, and

*Institute of Information and Computing Sciences, Utrecht University, Padualaan 14, 3584 CH Utrecht, The Netherlands. hansb@cs.uu.nl

†Department of Informatics, University of Bergen, N-5020 Bergen, Norway. telle@ii.uib.no

the 'text-book' modification to obtain an algorithm for the construction problem cannot be applied.

When the data for an algorithm do not fit into the main memory, but must be written to or obtained from secondary memory (e.g., a hard disk), then this has a severe impact on the time spent by the algorithm. Modern computers have large amounts of memory. Still, there are cases where the use of memory indeed still is an issue. For instance, if we allow a dynamic programming algorithm to run for several days, then the borderline between instances that can and cannot be handled often is determined by whether the data for the program fit into main memory. This happens for example for algorithms that use dynamic programming on graphs of bounded treewidth. These observations were the starting point for our investigations.

In this paper, we view DP as an algorithmic method applicable to a problem whose optimal solution for an input of size n can be found by combining optimal solutions to various subproblems. We associate with such a DP algorithm a directed acyclic graph D that will be central in our discussion. Each relevant subproblem is a node s of D, and represents a storage location for the solution value to this subproblem, which is typically a boolean value for a decision problem or a positive integer for an optimization problem. The solution value $v(s)$ stored at s is computed, as specified by the DP algorithm, by $v(s) := f(v(s_1), v(s_2), \ldots, v(s_k))$, for some specified function f and subproblems s_1, s_2, $\ldots$, s_k. In many cases the function f is a simple minimization or maximization, for example over sums of certain pairs of these values. The digraph D will have a directed edge $s_i s$ for each $1 \leq i \leq k$ representing the fact that the value stored at s_i is needed to compute the value stored at s. In a DP algorithm the resulting digraph D should have no directed cycles, *i.e.* it should be a dag, and it should have only a single sink, representing the full problem instance. In many formulations of DP the sink appears as a last maximization or minimization step over all entries in a certain range.

For the optimization (and decision) version of the problem, where we are simply asking for the solution value for the overall problem, we simply return $v(t)$, the value stored in the sink t of D. We define the space usage of this version of DP to be the maximum number of subproblem solution values kept in memory in course of the algorithm. The solution value to a subproblem s_i must therefore remain in memory until solutions to all subproblems s with an edge $s_i s$ have been computed. To arrive at the optimum space usage for a given DP algorithm on an input giving a digraph D we thus define $Space_{opt}(D)$ as the minimization of space usage over all topological sorts of D. Since we throw out the solution to a subproblem from memory as soon as allowed, the maximum number of storage locations, *i.e.* the space usage, for a particular topological sort $v_1, v_2, \ldots, v_m$ is the maximum over all $1 \leq i \leq m$ of $|\{v_j : j < i \wedge \exists v_j v_k \in E(D) \wedge k \geq i\}|$. Most textbooks on DP will mention this space-saving technique for a DP optimization problem.

We now turn to the construction version of DP, which is the subject of the current paper. In this version we need to find not only the value of an optimal solution, but also some object that achieves this value. Most textbooks mentioning DP will describe the following scheme for the construction version: first solve the optimization version, and then retrace from the sink of the dag D back through nodes that gave rise to this optimal

value. However, hardly any of these textbooks will mention that this requires us to store solutions to all subproblems, in case this subproblem is hit by the retracing, thus requiring $|V(D)|$ storage locations which is usually orders of magnitude higher than the $Space_{opt}(D)$ required for the optimization version.

The only fast space efficient construction versions of a DP application that we have found in the literature are for alignment problems in computational biology. Notably, a 1975 CACM-paper by Hirschberg [9] addresses this issue for the problem known as 'longest common subsequence', strongly related to 'string alignment'. The construction version is in this case solved with the same asymptotic time and space complexity as the optimization version. This result is based on a property of the problem which says that it can be broken into two parts where one corresponds to viewing the strings in reverse order. By combining the solutions to these two problems one finds a 'mid-way' alignment point, and can subsequently recurse on two sub-problems whose two digraphs have *combined size half* of the original digraph. This technique has become famous in the field of computational biology, and mentioned in textbooks, see *e.g.* [7], but is based on the problem-specific property mentioned above, and cannot be surmised from the digraph D.

This explains why a discussion of space-efficient construction versions for DP is difficult to carry out in a general setting, precisely because problem-specific properties may be necessary to get the best results. In this paper we primarily discuss two DP case studies in the field of graph problems: DP on a path decomposition and DP on a tree decomposition. For a graph with small treewidth the DP on its associated tree decomposition will allow the solution of various otherwise intractable graph problems, see *e.g.* [2, 6]. Various classes of graphs have small treewidth, for example the control-flow graphs of goto-free C programs have treewidth at most 6 [14, 5, 8]. Experiments and applications show that this is also useful in a practical setting. The algorithm of Lauritzen et al [11] to solve the probabilistic inference problem on probabilistic networks is the most commonly used algorithm for this problem and uses DP on a tree decomposition. Koster et al [10] used DP on tree decompositions to solve instances of frequency assignment problems. We show how to solve the construction variants of these problems while using only a negligible (twice) amount of more memory than for the optimization version, with a logarithmic factor increase in running time. The following list compares our result to the trade-off between time and space achievable by the previously best-known results.

	pre-1998	post-1998	Our result
Time	$O(n)$	$O(n^2)$	$O(n \log_p n)$
Space	n	$p \leq 2 \log n$	$2p$

Here Space is measured as the number of tables, each of size exponential in the treewidth but independent of n, that need to be stored simultaneously in memory for construction versions of dynamic programming on bounded treewidth graphs of n vertices. Since memory use in turn influences the wall-clock runtime of the programs, Space is the more important factor that we primarily focus on. The results quoted as pre-1998 and post-1998 are based on taking the fastest and most space-efficient optimization algorithms available at these dates, using the result of [3] that was presented in 1998, and turning these into construction

versions in a sensible way. The result of [3] provides us the minimum p such that one never needs to store more than p tables in memory. Our algorithm uses only p tables but the size of each table is doubled so Space is listed as $2p$. Notice that the logarithmic factor $\log_p n$ in the time use of our result grows smaller as p grows larger.

In the last section of the paper we discuss how our observations can be applied to the general case.

2 First Case Study: Dynamic Programming Using Path Decompositions

We start our exposition by an example of performing dynamic programming on a graph of bounded pathwidth, and describe our improved method in detail to ease its implementation for a particular application. A path decomposition of a graph $G = (V, E)$ is a sequence $X_1, \ldots, X_t$ of subsets of V, called *bags*, such that for all $1 \leq i_1 < i_2 < i_3 \leq t$, $X_{i_1} \cap X_{i_3} \subseteq X_{i_2}$; for all $\{v, w\} \in E$, there is an i, $1 \leq i \leq t$, $v, w \in X_i$, and $\bigcup_{1 \leq i \leq t} X_i = V$. The width of a path decomposition $X_1, \ldots, X_t$ is $\max_{1 \leq i \leq t} |X_i| - 1$, and the pathwidth of a graph is the minimum width of a path decomposition.

It is well known that many problems have a linear or polynomial time algorithm when restricted to graphs that have their pathwidth bounded by a constant, including many famous NP-complete graph problems. These algorithms use the DP paradigm. Going through the path decomposition from bag 1 to bag t, a table for the current bag is computed, using some 'local information', and the table for the preceding bag. The entries in a table for bag i generally express answers to subproblems for the graph induced by $X_1 \cup \cdots \cup X_i$, with special attention to what happens with the vertices in X_i in the solution at hand. Let us mention that correctness of these algorithms rely on the fact that any bag X_i is a separator of the graph, but the specific details of these algorithms are not of great importance to the exposition here.

We have a path decomposition $X_1, X_2, \ldots, X_t$ of an undirected graph G of pathwidth k, where t is linear in $|V(G)|$. For simplicity we assume that $|X_i| = k+1$ for each $1 \leq i \leq t$, and that the tables to be filled are $T_1[1..p(k)]$, $T_2[1..p(k)]$, $\ldots$,$T_t[1..p(k)]$. Thus, $T_i[j]$ should store the optimal value to a solution of the problem restricted to the graph induced by nodes $X_1 \cup X_2 \cup \cdots \cup X_i$, where the subgraph on nodes X_i are restricted to behave in a specified fashion as indicated by the table index j. The size $p(k)$ of the tables depends on the particular problem at hand, but for any problem which in general is NP-hard, it will be exponential in k. In the optimization version of the dynamic programming algorithm we keep only 2 tables in memory at any time, starting with tables T_1 and T_2, then T_2 and T_3, etc. In Figure 1 is a very simple example of a weighted graph G on $n = 6$ vertices and its path decomposition of width 2, for which we want to solve the maximum weighted independent set problem: In the 2-dimensional array below each column represents a table associated with a bag X_i of the path decomposition. In each bag we have ordered the vertices it contains. The leftmost column indicates the 8 indices of each table, *e.g.* index

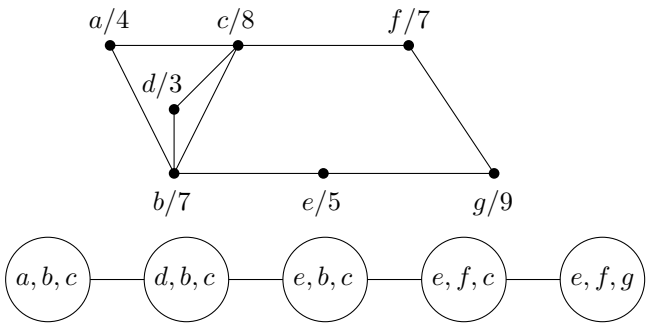

Figure 1: Example: graph G with tree decomposition

101 represents partial solutions where the first and third, but not the second, vertex in the bag should belong to the independent set. Thus, the 101 row has the entry $-\infty$ whenever the first and third vertex are adjacent, such as for the bag efg, since an independent set cannot contain two neighboring nodes. Vertices have been ordered so that they maintain the property of being first/second/third in every bag, note that this is always possible for a path decomposition where every bag has the same size, but not always possible for a tree decomposition. We have filled the tables with values as would be done by the optimization version of dynamic programming, storing also for each value the index of the previous column that gave rise to this value. If several such indices give the same value we just picked one arbitrarily. After solving the optimization version (the forward direction), the standard algorithm for the construction variant would then trace these pointers from the optimal entry at index 110 in the last table, back to the first, hitting the entries indicated by the * in those cells, giving the optimal solution $\{a, d, e, f\}$ with weight 19. Note that the sequence of dependency pointers, $p_1, p_2, \ldots, p_t$ (100, 100, 100, 110, 110 in the example) where $T_t[p_t]$ contains an optimal value for the overall solution on input graph G, and the value in $T_i[p_i]$ was computed based on $T_{i-1}[p_{i-1}]$, suffices to solve the construction variant. However, the backwards tracing as described here requires that all 5 tables, with pointers, are stored simultaneously.

	abc	dbc	ebc	efc	efg	efg w/ pointer to abc
000	0	4/100	7/100	7/000	7/000	7/100
001	8	8/001	8/001	8/001	17/001	17/001
010	7	7/010	7/010	14/000	14/010	14/100
011	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
100	4*	7/100*	12/100*	12/100	12/100	12/100
101	$-\infty$	$-\infty$	13/001	13/101	$-\infty$	$-\infty$
110	$-\infty$	$-\infty$	$-\infty$	19/100*	19/110*	19/100
111	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

We now describe an improved construction version which is space efficient at the expense of a logarithmic increase in running time. We will find $p_1, p_2, \ldots, p_t$ while storing only 2 tables

with pointers at any time, as follows: first find p_1 and p_t by computing in the forwards direction while storing pointers to the table T_1 only, *i.e.* computing T_i with its pointers to T_1 based only on T_{i-1} and *its* pointers to T_1. The final table will then be as in the rightmost column above, and we find $p_1 = 100$ and $p_5 = 110$.

Subsequently, we call subroutine FIND-INDEX(1,t), where FIND-INDEX(q,r) for $1 \leq q < r \leq t$, based on the knowledge of p_q and p_r, computes $p_{q+(r-q)/2}$, the dependency pointer for the table halfway between T_q and T_r. Since the only entry of table T_q needed to compute an optimal solution is the one with index p_q, and we do not need the actual value of an optimal solution, it will suffice to initialize $T_q[p_q]$ to 0 and all other entries of T_q to $-\infty$ (or $+\infty$ for a minimization problem). We then perform the forward direction of dynamic programming in the standard manner from T_q up to the halfway table $T_{q+(r-q)/2}$, and from here on to T_r storing pointers to this halfway table.

In the array below we have illustrated this procedure for the call FIND-INDEX(1,5) for the example, based on the knowledge that $p_1 = 100$ and $p_5 = 110$. The result of this call is the value $p_3 = 100$ which we find in $T_5[p_5]$. For this example the 2 calls FIND-INDEX(1,3) and FIND-INDEX(3,5) would subsequently compute the two remaining values p_2 and p_4.

	abc	dbc	ebc	efc	efg
000	$-\infty$	0	3	3/000	3/000
001	$-\infty$	$-\infty$	$-\infty$	$-\infty$	12/000
010	$-\infty$	$-\infty$	$-\infty$	10/000	10/000
011	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
100	0	3	8	8/100	8/100
101	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
110	$-\infty$	$-\infty$	$-\infty$	15/100	15/100
111	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

In general, the call FIND-INDEX(q,r) takes time $O(r - q)$ while storing 2 tables with pointers, and computes a single optimal pointer. A balanced divide-and-conquer strategy will therefore compute all pointers in time $O(n \log_2 n)$ while storing 2 tables.

This technique works with many problems restricted to graphs with bounded pathwidth, for instance for the construction variants of problems that can be formulated in (extended) monadic second order logic.

Lemma 1 *Let k be a constant, and let $Q(S)$ be a monadic second order logic formula with S a free vertex or edge set variable. Each of the following problems can be solved in $O(n \log n)$ time with $O(1)$ additional memory for graphs G, given together with a path decomposition of width at most k: find a (vertex or edge) set S such that $Q(S)$ holds in G; find a set S such that $|S|$ is minimum (maximum) over all sets such that $Q(S)$ holds in G.*

Similar results can be obtained for other types of problems, e.g., for cases where vertices and/or edges have weights. The result can be strengthened however, as we will do in the next section.

3 Second Case Study: Dynamic Programming Using Tree Decompositions

The essence of the technique applied in the previous example, viewed in terms of the directed path $X_1, X_2, \ldots, X_t$ of bags, was that we found a 'small and good' separator of this directed path (consisting of the table at the half-point). We thus managed to solve the construction version by breaking this problem in two halves recursively. In our next case study, DP on graphs of bounded treewidth, the situation is generalized to graphs with a tree structure, but the general idea remains the same, find a small and good separator, and break the problem into the parts that result from removing the separator.

A tree decomposition of a graph $G = (V, E)$ is a pair (T, X) with $T = (I, F)$ a tree, and $X = \{X_i \mid i \in I\}$ a family of subsets of V, such that for all nodes i_1, i_2, $i_3 \in I$, if i_2 is on the path from i_1 to i_3 in T, then $X_{i_1} \cap X_{i_3} \subseteq X_{i_2}$; for all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$, and $\bigcup_{i \in I} X_i = V$. We call the sets X_i the *bags*. The *width* of (T, X) is $\max_{i \in I} |X_i| - 1$, and the treewidth of a graph G is the minimum width of a tree decomposition of G. Like pathwidth, treewidth has nice algorithmic properties, and allows many problems that are hard on general graphs to be solved in linear or polynomial time when restricted to graphs of bounded treewidth; see e.g., [2, 6, 10]. These algorithms are again usually of dynamic programming type and have the following form. A node of T is chosen as root. For each node i, we compute a table. These tables are computed in bottom up order: to compute a table, we use the information of the tables of the children, plus some 'local information' about the subgraph induced by bag X_i. The decision problem can be solved once the table of the root is known. For the construction problem, a corresponding solution can be constructed by going downwards from root to leaves using the information in the tables. We consider now one illustrative example.

Assume we have a tree decomposition (T, X) of a graph G of treewidth k. For an optimization version of DP, for example answering the question 'What is the size of the largest independent set in G?', the information contained in the table at a child node of T is superfluous once the table of its parent has been updated. Since the size of tables is exponential in k, it is in practice very important to carefully re-utilize these memory locations in order to minimize time-consuming I/O to external memory. A simple linear-time algorithm will in a pre-processing step find a bottom-up traversal of T that minimizes the number of tables stored at any time during the dynamic programming. This number lies between the pathwidth of T and twice the pathwidth of T [3].

The construction variant of DP on tree decompositions, of the form 'Find a largest independent set in G', can be solved in the standard manner, by first performing the 'forward' direction that finds an optimal value, and then tracing an optimal entry in the final table (the root) back through those entries in earlier tables that gave rise to it (down to the leaves). However, since tables are no longer superfluous, the number of tables to be stored under this scheme is linear in the size of G. This idea forms the pre-1998 entry in the comparison given in the Introduction and is clearly impractical. The post-1998 entry is based on the result of [3] (presented first at SWAT'98) which showed that the

optimization version could be solved using at most $p < 2 \log n$ tables, and repeats this optimization version once for each of the n bags. Our current aim is to solve the construction variant as fast as possible under the practically oriented constraint that we store only an asymptotically optimal number of tables, *i.e.* linear in the pathwidth of T.

We use a method similar to that of the previous section. For the best tradeoff, we split the tree in several parts, using the following simple lemma.

Lemma 2 *For any $q \geq 0$ and every tree T on n vertices there is a set $S \subseteq V(T)$ with $|S| \leq 2^{q+1} - 1$ such that every connected component of $T - S$ is of size at most $\lceil n/2^q \rceil$.*

Proof. We proceed by induction on q. It is well known that every tree T on n vertices has a *centroid*, i.e. a vertex v such that each connected component of $T - v$ is of size $\leq \lceil n/2 \rceil$. Let $q \geq 1$ and suppose that for all $l \leq q$ the lemma holds. Then there is a set S containing at most $2^{q+1} - 1$ vertices such that every connected component of $T - S$ has size $\leq \lceil n/2^q \rceil$. At most 2^{q+1} of these components have size $> \lceil n/2^{q+1} \rceil$. Each of these large components has size $\leq \lceil n/2^q \rceil$ and they have themselves a centroid. Thus by removing at most 2^{q+1} such centroid vertices one can split all large components into components of size $\leq \lceil n/2^{q+1} \rceil$. The number of vertices one needs to remove is $\leq |S| + 2^{q+1} \leq 2^{q+1} + 2^{q+1} - 1 = 2^{q+2} - 1$. ∎

This result allows us to choose the factor q specifying the size of components in the recursive step. This means that we can choose a function $f : \mathbf{N} \to \mathbf{N}$ that will guide the running time of the divide-and-conquer strategy, as follows. Find the smallest q such that $f(n) \leq 2^q$. Pick the $O(2^q)$ special vertices guaranteed by Lemma 2, do the dynamic programming starting from the leaves of T and when reaching a special vertex x start recording pointers to the table of x until reaching the next special vertex y on the path to the root. Store all pointers from y to x until done with this round, and start recording pointers now to y. When done with this round the stored pointers will give the optimal indices for all these special vertices. In the recursion all components are small, and simply use the trick of pre-initializing the optimal indices in the old special vertices to 0 and its other indices to +/- ∞. Note that the old special vertices are now leaves and the process can be repeated. This gives us the following result.

Theorem 1 *Let k be a constant, let $f : \mathbf{N} \to \mathbf{N}$ be a function, and let $Q(S)$ be a monadic second order logic formula with S a free vertex or edge set variable. Each of the following problems can be solved in $O(n \log_{f(n)} n)$ time with $O(f(n)+p)$ additional memory for graphs G, given together with a tree decomposition (T, X) of width at most k, with p the pathwidth of T: find a (vertex or edge) set S such that $Q(S)$ holds in G; find a set S such that $|S|$ is minimum (maximum) over all sets such that $Q(S)$ holds in G.*

We remark that the result also holds for other problems, as long as the algorithm on the tree decomposition has the form sketched above. If the maximum table size is $g(k, n)$, k the treewidth and n the number of vertices, then the time for the construction version is $O(n \log_{f(n)} n)$, and the memory use is $O(g(k, n) \cdot (f(n) + p))$, p the pathwidth of the tree T.

Note that by the results from [3], the maximum number of tables that the decision algorithm needs to have stored in memory is $\Theta(p)$.

Corollary 3 *Setting $f(n)$ to the pathwidth p of T, we solve the construction version in time $O(n \log_p n)$, using about twice as much memory as for the optimization version.*

The factor two for the memory derives from the need to store the indices to previous tables, so in fact the number of tables remain the same, but each table index stores now two pieces of information instead of just one. This corollary is nice since when we need much memory for the optimization/decision version, due to the tree in the tree decomposition having high pathwidth, we get a smaller extra factor $\log_p n$ for the construction version.

4 Generalizing from the Case Studies

Let us now look at the general case of DP. Described intuitively, the construction version of DP is usually solved by first solving the optimization version by moving in a forward direction in the associated DP dag D, and then retracing from the sink of the dag D back through nodes that gave rise to the optimal value. We give an inductive definition of the nodes in the 'optimal subgraph' retraced in this second step: It contains the sink node t of D. Inductively, if it contains a non-source node whose value was computed by $f(v(s_1), v(s_2), ..., v(s_k))$, then it will also contain at least one of $s_1, s_2, ..., s_k$, but precisely how many of them it contains depends on f. For example, if f is simply a minimization then we may choose arbitrarily one of $s_1, s_2, ..., s_k$ whose solution is smallest, or if f is minimization over sums of certain pairs, then we choose a corresponding pair from $s_1, s_2, ..., s_k$ with smallest sum and include both of them. We note that in the DP solutions to longest common subsequence and other sequence alignment problems, a single node from $s_1, ..., s_k$ is chosen at each step, and the nodes in the 'optimal subgraph' thus form a path.

In the path decomposition example, there would be a node of the dag D for each entry $T_i[j]$ of a table, with an arc from $T_i[j]$ to $T_{i+1}[j']$ for some $j' > j$ if the entry at j' was computed by an optimization over a range of entries that included j. Usually, the optimization is a maximization or minimization over single entries, and not e.g. sums of pairs of entries, but the specific arcs will vary from problem to problem. In addition we would have a sink node t with incoming arcs from each node associated with the last table. In the case that the optimization for every table entry is over single entries, the optimal subgraph is again a path. But of course, the digraph D itself is not a path. However, the path decomposition imposed a path structure on D, of width k, and our technique for achieving space usage of 2 tables at the expense of an extra logarithmic factor in the runtime relied heavily on this path structure. In the tree decomposition example, we had a similar situation where although the DP dag D is not a tree it had tree-structure imposed on it by a tree decomposition, and our space-efficient technique relied on this tree-structure.

What can we learn in general from the two case studies? The core of the general technique is to find a small and good separator of the dag D, and break the problem into the parts that result from removing the separator. The path decomposition (and

tree decomposition) (T, X) of G was computed from the graph G, and the resulting DP dag $D_{(T,X)}$ was then defined, based on the problem at hand and on (T, X). The clue to our space-saving techniques was that $D_{(T,X)}$ inherited the path (or tree) structure. In the general DP case we are handed the dag D without recourse to a path or tree decomposition. How then to apply the same technique of imposing a path (or tree) structure on the dag D?

The right theoretical tool here is the concept of strong directed treewidth. Strong treewidth, as defined by Seese in [12], is related to a strong tree decomposition. These differ from a tree decomposition in that each vertex appears in exactly one bag (the bags are thus a partitioning of the graph vertices), and edges of the graph are now allowed to go between vertices in adjacent bags, and not necessarily in the same bag. Since our dag is directed we do not allow the bag containing t to be the child of the bag containing s if we have an edge from vertex s to t, thus no edges point downwards. As usual, the goal is to minimize the size of the largest bag. With this definition, the tree structure naturally induced by T on the digraph $D_{(T,X)}$ is a strong tree decomposition of $D_{(T,X)}$. Thus, to generalize our technique we use a good strong directed tree decomposition of the digraph D.

The digraph D is oftentimes too large to be kept in memory, and the practical approach to finding fast construction variants in this case must rely on problem-specific properties. Here it is helpful to note that the digraph D for a DP algorithm is not some arbitrary digraph, but arises from some fairly clear structural aspects of the problem at hand, by defining the value of an optimal solution recursively in terms of the optimal solutions to subproblems. This (simple) recursive definition will also define the structure of the digraph D, and a starting point is to look for separator theorems like our Lemma 2 for graphs with this structure. Such separator theorems are well-known for various classes of graphs, like planar graphs, and in particular for most graph classes with a recursive structural definition. On a case-by-case basis one can then, depending on the separator theorems available for the particular digraph, develop fast and space-efficient construction algorithms similar to the ones given here.

Acknowledgement

Thanks to Fedor Fomin for fruitful discussions on this topic.

References

[1] J. Alber, H. L. Bodlaender, H. Fernau, T.Kloks, R. Niedermeier: Fixed Parameter Algorithms for Dominating Set and related problems on Planar Graphs. *Algorithmica* Vol. 33, 2002, 461–493.

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12, 308–340 (1991).

[3] B. Aspvall, A. Proskurowski, J. A. Telle, Memory requirements for table computations in partial k-tree algorithms, *Algorithmica*, Special issue on Treewidth, Graph Minors and Algorithms, H. Bodlaender eds., Vol. 27, Number 3, 2000, 382–394.

[4] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25, 1305–1317 (1996).

[5] H. Bodlaender, J. Gustedt, J. A. Telle, Linear-time register allocation for a fixed number of registers, *Proceedings SODA'98*, 574–583, San Fransisco, USA.

[6] H. L. Bodlaender, Treewidth: Algorithmic Techniques and Results. *Proceedings MFCS'97,* LNCS 1295:29–36.

[7] D. Gusfield, Algorithms on strings, trees, and sequences, Cambridge University Press (1997).

[8] J. Gustedt, O. Mæhle, J. A. Telle, The Treewidth of Java Programs, *Proceedings ALENEX'02 — 4th Workshop on Algorithm Engineering and Experiments*, San Francisco, January 4-5, 2002, LNCS 2409: 86–97.

[9] D. S. Hirschberg, A linear space algorithm for computing longest common subsequences, *Commun. Assoc. Comput.Mach.*, 18, 341–343 (1975).

[10] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40, 170–180 (2002).

[11] S.J.Lauritzen, D.J.Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems, *The Journal of The Royal Statistical Society. Series B (Methodological)*. 50:157-224, 1988.

[12] D. Seese. Tree-partite graphs and the complexity of algorithms. In L. Budach, editor, *Proc. 1985 Int. Conf. on Fundamentals of Computation Theory, Lecture Notes in Computer Science 199*, pages 412–421, Berlin, 1985. Springer Verlag.

[13] D. Seese, F. Schlottmann, Logical Ways to Reduce Complexity by Avoiding Grids: A Survey on Results and Problems, manuscript (2002)

[14] M. Thorup, Structured Programs have Small Tree-Width and Good Register Allocation, *Information and Computation* 142, 159–181, 1998.