Memory requirements for table computations in partial k-tree algorithms

Bengt Aspvall
University of Bergen
Bergen, Norway

Andrzej Proskurowski
University of Oregon
Eugene, Oregon, USA

Jan Arne Telle
University of Bergen
Bergen, Norway

Abstract

This paper addresses memory requirement issues arising in implementations of algorithms on graphs of bounded treewidth. Such dynamic programming algorithms require a large data table for each vertex of a tree-decomposition T of the input graph. We give a linear-time algorithm that finds the traversal order of T minimizing the number of tables stored simultaneously. We show that this minimum value is lower-bounded by the pathwidth of T plus one, and upper bounded by twice the pathwidth of T plus one. We also give a linear-time algorithm finding the depth-first traversal order minimizing the sum of the sizes of tables stored simultaneously.

1 Introduction

Many NP-hard graph problems have linear-time algorithms when restricted to graphs of treewidth bounded by k (equivalently, partial k-trees), for fixed values of k. These algorithms have two stages: the first stage finds a bounded width tree-decomposition of the input graph and is followed by a dynamic programming stage that solves the problem in a bottom-up traversal of that tree. Theoretically, the first stage has linear-time complexity (see [3]), but no general practical implementation for treewidth $k \geq 5$ exists. In this paper we focus on practical implementations of the second stage.

Dynamic programming algorithms on bounded treewidth graphs have been studied since the mid-1980s leading to several powerful approaches, see [4] for an overview. Recent efforts have been aimed at making these theoretically efficient algorithms amenable also to practical applications [13, 14]. Experimental results so far have

[1]An extended abstract of this paper will be presented at SWAT'98 - Sixth Scandinavian Workshop on Algorithm Theory in Stockholm, Sweden, July 1998.

not been negative: for example, using a 150 MHz alpha processor-based Digital computer, the maximum independent set problem is solved for treewidth-10 graphs of 1000 vertices in less than 10 seconds [7]. The given tree-decomposition for this particular example had about 1000 nodes and associated with each node is a data table of 2^{11} 32-bit words, so without reuse of data tables the combined storage would exceed 8 Megabytes of memory. With our algorithms we need only 80 Kilobytes. These initial investigations already show that avoidance of time-consuming I/O to external memory is an important issue (see [5] for a good overview of issues related to external memory use in graph algorithms and the expected increased significance of these issues in coming years).

In the bottom-up traversal of the tree-decomposition that accompanies the dynamic programming stage, we are free to choose both the root and the traversal order. Moreover, the information contained in the table at a child node is superfluous once the table of its parent has been updated, and the table of the parent need not be created until it is ready to be updated by the table of the first child. Given a tree, a natural question is how to find a good root and a good traversal order to minimize the table requirement, *i.e.*, the minimum number of tables that need to be stored simultaneously, under the assumption that all tables will be kept in the main memory. In the next section we answer this question and give a careful description of the resulting algorithm to ease the task of implementation. In Section 3, we show an interesting relationship between the table requirement of a tree T and the pathwidth of T, giving asymptotically tight bounds relating these two parameters. In the case when the size of the table is not the same over all nodes of the tree-decomposition, we may ask instead for a traversal order that minimizes the sum of sizes of tables that need to be stored simultaneously. In Section 4 we give a practical linear-time algorithm answering this question for depth-first search (dfs) traversal orders. The traversal order output by the algorithm in Section 2 minimizing table requirement is indeed a dfs order, and dfs orders have the advantage of being easy to implement. We conclude in Section 5 with a discussion of future research.

2 Table requirement

In the following, we will refer not to the bounded treewidth graph G itself but instead focus on the adjacencies of its tree-decomposition T. The interested reader may see, *e.g.*, [4] for definitions related to bounded treewidth graphs. We assume that in order to solve a discrete optimization problem on G, some function on T has to be computed. The value of this function is independent of the choice of root for the tree, and is obtained from the completed computation of the table associated with the vertex chosen as root. The computation of this final table is a bottom-up process which involves computation of tables for all other vertices of the tree.

The table of a leaf vertex is defined by the base case of the function. The table of a non-leaf vertex is updated according to the contents of the completed table of its child vertex, after which that latter table can be discarded. This table update operation requires the simultaneous presence of the parent and child tables. The table at a vertex is completed once it has been updated using the contents of the tables of all its children. We must thus pay for storage of the table at a vertex u from the moment of its creation, allow the table to become completed by being updated by completed tables of all children of u, update the table of the parent of u, and only then discard the table of u. We summarize the above discussion in a formal definition:

Definition 2.1 *An ordering of edges $e_1, ..., e_{n-1}$ of a rooted tree with $n > 1$ vertices is a bottom-up traversal if for any edge $e_j = (u, parent(u))$, all edges in the subtree rooted at u have indices less than j. The* lifetime *of the table of vertex u in this traversal is the interval $[i, j]$ where e_i is the lowest-indexed edge containing u and e_j is the highest-indexed edge containing u. The table requirement of this traversal order is the maximum number of tables alive at any time,* i.e. $\max_{1 \leq i < n} |\{v_k : i \in lifetime(v_k)\}|$.

For adjacent vertices u and v of T we define $tabreq(u, v)$ as the minimum table requirement of the subtree rooted at u with overall root chosen so that u has parent v, taken over all possible bottom-up traversals of this subtree.

Theorem 2.2 *In a given rooted tree, let a non-root vertex u have the parent v and let x and y be the children of u (if any) such that $tabreq(x, u) \geq tabreq(y, u) \geq tabreq(w, u)$ for any other child w of u (if u has only one child, define $tabreq(y, u)$ to be 0.) Then, the table requirement to compute the table associated with u is given by*

$$tabreq(u, v) = \begin{cases} 1 & \text{for a leaf } u \\ \max\{tabreq(x, u), 2, tabreq(y, u) + 1\} & \text{for a non-leaf } u. \end{cases} \tag{1}$$

Proof. The base case of a leaf u follows from the definition of $tabreq(u, v)$. We continue with the non-leaf case. That the value given by (1) suffices follows from noting the following: (i) First compute recursively the table of the child x with largest table requirement; this requires at most $tabreq(x, u)$. (ii) Then create the table of u and update it with the table of x before discarding table of x; this requires at most 2 tables. (iii) Finally compute recursively tables of remaining children of u one by one, in any order, updating the table of u before discarding the table of the child; this requires at most $tabreq(y, u) + 1$. That the value given by (1) is necessary follows by noting: (i) To compute the table of a non-leaf vertex u, it is necessary to have computed the tables of all its children; this requires at least $tabreq(x, u)$. (ii) To update the table of u with the table of a child, both these tables must be present in memory; this requires at least 2 tables. (iii) After a table has been created, it must be stored at least until the table of its parent has been created. Thus, only for a single child c of u will it be possible to compute its table while storing only tables that belong to the subtree rooted at c. This entails a requirement of at least $tabreq(y, u) + 1$ (achieved by choosing $c = x$) which is minimal since $tabreq(x, u) \geq tabreq(y, u)$. ∎

For a tree T and a vertex r in T, we denote by T_r the rooted version of T resulting from designating r as its root. Since r does not have a parent in T_r, we denote the table requirement of T_r by $tabreq(T_r)$. This value is the minimum number of tables necessary to complete the table at r, and hence to compute the function on T_r solving the original problem for G. When the tree is rooted at u, the value of $tabreq(T_u)$ is given by the right-hand side of (1).

Corollary 2.3 *The right-hand side of (1) gives the value of $tabreq(T_u)$; in this case, all neighbors of u are considered in defining x and y.*

Corollary 2.4 *For an n-vertex tree T rooted in an arbitrary vertex u we have*

$$tabreq(T_u) \leq \lfloor \log_2 \frac{4}{3}(n+1) \rfloor$$

Proof. Let $S(i)$ be the number of nodes in the smallest rooted tree with table requirement i. By Theorem 2.2 we have the recurrence $S(1) = 1$, $S(2) = 2$ and $S(i) = 2S(i-1) + 1$ for $i \geq 3$. The solution to this recurrence is $S(i) = 3 \times 2^{i-2} - 1$ for $i \geq 3$. Rearranging and taking logarithms the result follows. ∎

Note that $tabreq(T_u)$ will usually differ from $tabreq(T_x)$ for $x \neq u$. We define the table requirement of a tree T as the minimum value of $tabreq(T_u)$ over all vertices u of T. This parameter is of interest also because of its similarity to graph searching games (see, for instance, [6]), to the minimum stack traversal problem [1] and to the pathwidth parameter, discussed in the next section. Indeed, the algorithm we now present to compute this parameter uses a traversal strategy resembling closely that of [1] as well as those used by [12, 9] to compute the pathwidth of a tree.

The algorithm will consist of a single bottom-up phase, which will start at the degree one vertices of the tree (leaves), and end in a root r with minimum value of $tabreq(T_r)$ (over all vertices of T). For each vertex v, we keep track of $larg(v)$ and $next.larg(v)$, the two largest table requirements reported by neighbors of v. We also maintain an array S of stacks to guide the order of processing vertices of T. The stack $S[i]$ contains those leaf vertices of the remaining tree of unprocessed vertices whose subtrees have table requirements of value i. Vertices are popped from the non-empty stack $S[i]$ with the smallest value of i. The algorithm, which assumes that T has at least two nodes, can now be described as follows (see Figure 1 for an example of algorithm execution):

1 Set $i = 1$. Push all degree one vertices of T on $S[1]$. For all vertices v, set $larg(v) = next.larg(v) = 0$.

2 While T has more than one vertex remaining

2a While $S[i]$ is empty increment i.

2b Pop a vertex w from $S[i]$. Let v be the single neighbor of w (its parent) in the tree T of unprocessed vertices.

2c Report the value i to v, update the value of $larg(v)$ to $\max\{i, larg(v)\}$, and also update $next.larg(v)$ to $\max\{next.larg(v), \min(larg(v), i)\}$.

2d If v has received reports from all but one neighbor then push v on the stack $S[j]$, $j = \max\{larg(v), 1 + next.larg(v), 2\}$. Remove w from T.

3 Report the only remaining vertex r in T as an optimal root, with table requirement j such that $r \in S[j]$.

Theorem 2.5 *The node r of T which remains unprocessed upon termination of the algorithm (when the conditional in the outer while loop is false) is an optimal root for minimizing table requirement of T.*

Proof. (By contradiction.) Assume that an optimal root $r^* \neq r$ has value $m^* = tabreq(T_{r^*}) < tabreq(T_r)$. Let r have the neighbors $w, w_1, ..., w_k$ with w on the path from r to r^* in T and let $tabreq(w_1, r) \geq tabreq(w_2, r) \geq ... \geq tabreq(w_k, r)$. Table requirements are defined by Theorem 2.2. We have $tabreq(w_1, r) \leq tabreq(r, w) \leq m^*$ and $tabreq(w_2, r)+1 \leq tabreq(r, w) \leq m^*$. We also have $tabreq(w, r) \leq m^*$ since otherwise, considering the previous statement, r would be on a lower-indexed stack than w and r would not be processed last. Since we assumed $m^* < tabreq(T_r)$ we have w and w_1 the two neighbors of r with largest table requirements $tabreq(w, r) = tabreq(w_1, r) = m^* > tabreq(w_2, r)$. The two vertices, w and w_1, are thus the last neighbors of r processed by the algorithm, with the parameter $i = m^*$. But this implies that the penultimate neighbor of r processed by the algorithm causes r to be pushed on the stack $S[m^*]$ in step 2d. Thus, r would be popped next and not remain as the last unprocessed vertex. ■

Using the observation that $tabreq(T_r)$ increases over its subtrees' requirement only when $larg(v) = next.larg(v)$, we arrive at the following result:

Corollary 2.6 $tabreq(T_r) - 1 \leq tabreq(T_{r^*}) = m^*$*, for any vertex r and an optimal root r^* in a tree T.*

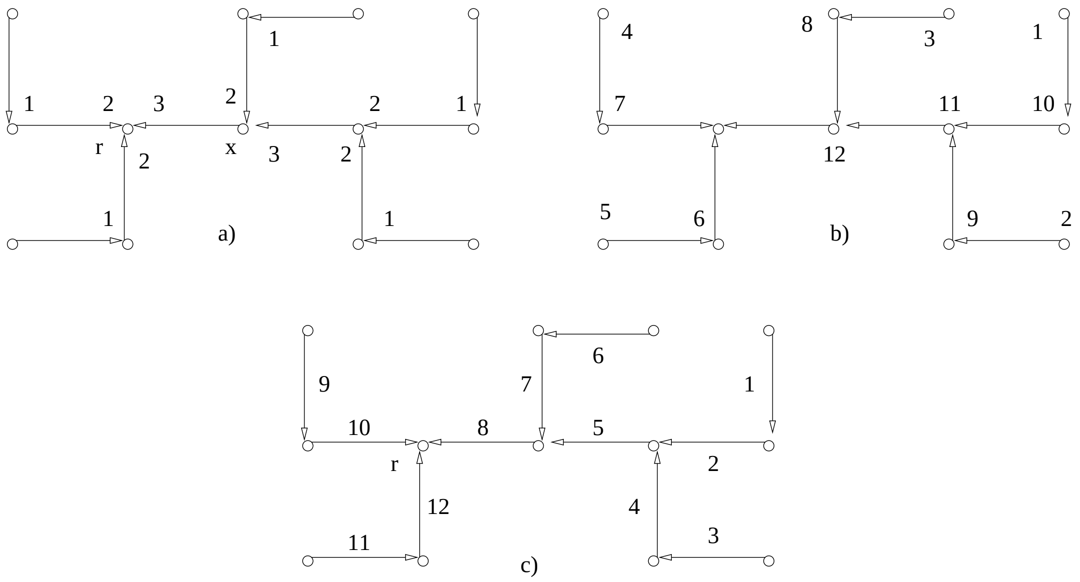

Figure 1: a) A tree with its optimal root r and values of table requirement for subtrees, as computed by our algorithm. Optimal root r gives table requirement 3, while rooting in x would give table requirement 4. b) The order in which vertices were popped from the stack during computation of optimal root. c) An optimal traversal order, as defined by the order of execution of update-table$(u, parent(u))$ in call of DFS(r).

Proof. Let P be the unique path $r^* = r_1, r_2, ..., r_d = r$ between r^* and r. Let C_i be the neighbors of r_i that are not on P. The subtree rooted in a given vertex x of C_i does not depend on the choice of overall root r or r^*. Thus in both cases x reports the same memory requirement, of value at most m^*, to its parent r_i. By Theorem 2.2, a report of table requirement of value $m^* + 1$ or more occurs only if one child reports at least $m^* + 1$ or at least two children report m^*. Thus, in T_r such reports only occur on the path P, from some r_i to its parent $r_{i+1}, i \geq 1$, but in no other place. We conclude that $tabreq(T_r) \leq m^* + 1$. ∎

The values $\max\{larg(v), 1 + next.larg(v), 2\}$, computed in step 2c by the algorithm for each vertex v, gives the table requirement for the subtree of T_r rooted at v. To find a traversal order minimizing table requirement let a node v in T_r have children $child_1(v), ..., child_{c_v}(v)$, with $child_1(v)$ having the largest table requirement over all subtrees of T_r rooted at these vertices. A call of the procedure DFS(r) shown below will then perform the dynamic programming computation on T while minimizing the number of tables stored simultaneously:

```
DFS(v)
if v a leaf then create-table(v);
else {
    DFS(child_1(v)); /* The most expensive subtree first */
    create-table(v);
    update-table(child_1(v), v);
    discard-table(child_1(v));
    for i=2 to c_v{
        DFS(child_i(v));
        update-table(child_i(v), v);
        discard-table(child_i(v));}}
```

This strategy results in a traversal order minimizing table requirement as described in the proof of Theorem 2.2.

3 Pathwidth and table requirement

Pathwidth is a graph parameter whose definition is closely related to treewidth [11].

Definition 3.1 *A path-decomposition of a graph G is a sequence $X_1, ..., X_m$ of bags, which are subsets of the vertex set $V(G)$ of G, such that for each edge of G there is a bag containing both its end-vertices and the bags containing any given vertex form a connected subsequence. G has* pathwidth *at most k, $pw(G) \leq k$, if it has a path-decomposition where the cardinality of any bag is at most $k+1$.*

As an aside, we remark that the definition of treewidth is similar, except that the bags are nodes of a tree, as opposed to being nodes of a path (sequence), and the bags containing a given vertex induce a connected subtree. There is a linear-time algorithm computing pathwidth and path-decomposition of a tree, [9], which uses a traversal similar to our minimum table requirement algorithm, based on the following Theorem:

Theorem 3.2 *[12] For a tree T, $pw(T) > k$ if and only if there exists a vertex v in T such that $T \setminus \{v\}$ has at least three components with pathwidth at least k.*

We will use the following corollary to Theorem 3.2:

Corollary 3.3 *In a tree T with $pw(T) = k+1$, either there is a vertex x s.t. all components of $T \setminus \{x\}$ have pathwidth at most k, or the set of edges e, such that deleting e from T gives two trees both of pathwidth $k+1$, induce a path $P = e_1, ..., e_p$, with $p \geq 1$.*

We will now show a close relation between the two parameters pathwidth and table requirement for a tree.

Theorem 3.4 *Any tree T has a node r such that*

$$pw(T) + 1 \leq tabreq(T_r) \leq 2pw(T) + 1$$

Proof. We show that the first inequality holds even for an arbitrary choice of r. Consider any bottom-up traversal order of T_r, $e_1, e_2, ..., e_{n-1}$ that defines the order of table updates. Let $e_i = (v_i, parent(v_i))$ and define $X_1 = \{v_1, parent(v_1)\}$ and $X_i = X_{i-1} \setminus \{v_{i-1}\} \cup \{v_i, parent(v_i)\}$ for $2 \leq i \leq n-1$. Then $X_1, ..., X_{n-1}$ is a path-decomposition of T as each edge is contained in a bag and, by construction, the bags containing a given vertex are consecutive. Moreover, the maximum cardinality of a bag is the memory requirement of this particular bottom-up traversal of T_r, as the table at a node is discarded precisely after the table of the parent of the node has been updated. Thus $pw(T) + 1 \leq tabreq(T_r)$.

We show the second inequality by induction on $pw(T)$. The base case is $pw(T) = 0$ with T being a single node r, and $tabreq(T_r) = 1$ as a single table suffices. For the inductive step, let us assume the second inequality for trees with pathwidth k and let $pw(T) = k + 1$. We must show that table requirement of T is at most $2k + 3$. Apply Corollary 3.3 to T and choose as root of T a vertex x as described by the Corollary, if it exists, or else consider the path $P = e_1, ..., e_p$ as described by the Corollary, with $e_i = \{v_{i-1}, v_i\}, 1 \leq i \leq p$ and choose v_p as the root. In the former case, the inductive assumption and Corollary 2.6, together with Theorem 2.2, gives us $tabreq(T_x) \leq 2k + 3 \leq 2pw(T) + 1$. In the latter case, note that the endpoint v_0 of P, the lowest node of P in the tree T_{v_p}, by Corollary 3.3 has all its subtrees of pathwidth at most k. Hence by the inductive assumption and Corollary 2.6 these children report table requirement at most $2k + 2$ to v_0, so that by Theorem 2.2 v_0 reports table requirement at most $2k + 3$ to its parent v_1. By induction on i, it follows that each node $v_i, i \geq 1$ on P receives report of at most $2k + 3$ from its child v_{i-1} and a report of at most $2k + 2$ from all its other children (since by Corollary 3.3 the subtrees rooted in these other children must have pathwidth at most k). By Theorem 2.2 we conclude that $tabreq(T_r) \leq 2k + 3 = 2(k + 1) + 1 = 2pw(T) + 1$. ∎

It is easy to check that for a complete ternary tree T with k levels and root r we have $pw(T) + 1 = tabreq(T_r) = k$.

We show that also the second bound in Theorem 3.4 is asymptotically tight, by giving a class of trees $G^k, k = 1, 2, ...$ with $pw(G^k) = k$ and minimum table requirement $2k$, denoted $tabreq(G^k) = 2k$. We define G^k recursively, with G^1 having two nodes, one of which is designated as its center, c_1, and a single edge. The general tree G^k consists of six copies of G^{k-1} whose centers (isomorphic to c_{k-1}) have pairwise common neighbors, x, y, z, all adjacent to the center c_k of G^k. (See Figure 2 for an illustration of the definition of G^k.)

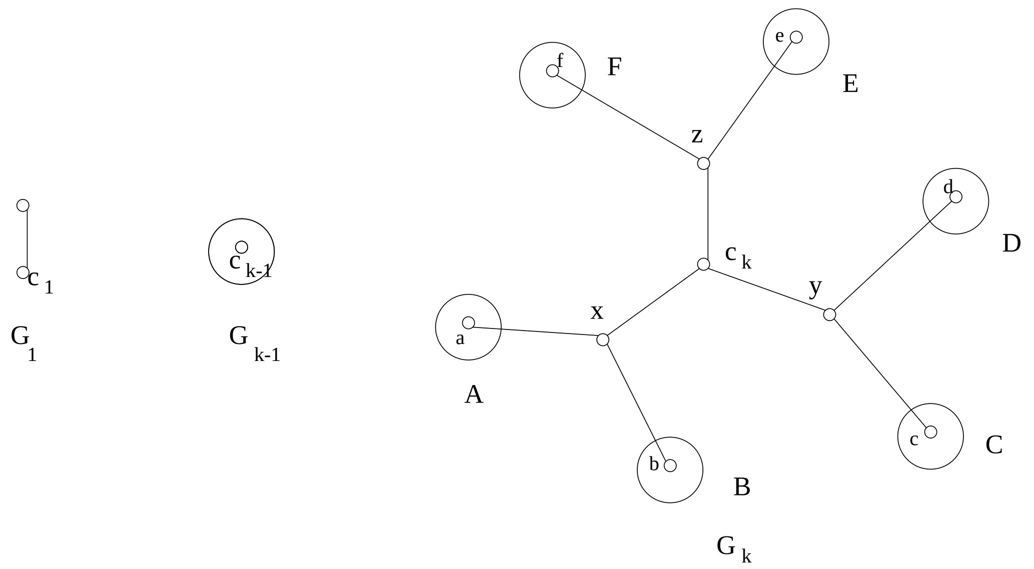

Figure 2: The recursive construction of G^k, with G^1, G^{k-1}, and G^k constructed from 6 copies of G^{k-1} attached to the central node of G^k labelled c_k via nodes x, y and z.

Fact 3.5 *$pw(G^k) \leq k$ and $tabreq(G^k) \geq 2k$*

Proof. By induction on k, we show that G^k has a path-decomposition $X_1, ..., X_m$ with $c_k \in X_m$ (or $c_k \in X_1$ by a reversal), where the cardinality of any bag is at most $k+1$. The graph G^1 is an edge and the two end-points make up the only bag. Let A, B, C, D, E, F be the 6 copies of G^{k-1} in G^k, as in Figure 2, and let $P_A, .., P_F$ be path-decompositions of A, B, C, D, E, F, respectively, with bags of cardinality at most k and with each central node, a, b, c, d, e, f, in the highest-numbered bag (such path-decompositions exist by the inductive assumption.) A path-decomposition of G^k with c_k in the highest-numbered bag can be constructed as follows:

$$\begin{aligned} &P_A, \{x, a\}, P_B \cup \{x\}, \{x, b, c_k\}, P_C \cup \{c_k\}, \{c, c_k, y\}, \{c_k, y, d\}, \\ &\quad Rev(P_D) \cup \{c_k\}, P_E \cup \{c_k\}, \{e, z, c_k\}, \{z, c_k, f\}, P_F \cup \{c_k\} \end{aligned} \tag{2}$$

where $Rev(P_D)$ is the reversal of P_D (having d in the lowest-numbered bag) and, for a path-decomposition S and node s, $S \cup \{s\}$ denotes the path-decomposition obtained by augmented each bag of S by s. The path-decomposition is read from left to right where, *e.g.*, $P_A, \{x, a\}, P_B \cup \{x\}$ indicates that the bags of path-decomposition for A are followed by the bag $\{x, a\}$ which is followed by the bags of path-decomposition for B, each augmented by the node x. Thus Equation 2 is readily seen to be a path-decomposition of G^k, $k \geq 2$, and no bag has size more than $k+1$. Thus, $pw(G^k) \leq k$. We show the second statement in the fact also by induction on k. The graph G^1 is an edge with pathwidth 1 and table requirement 2. Whatever root

r we pick in G^k, its central node c_k will have two subtrees, each in turn with two subtrees each with table requirements $2(k-1)$, by induction. Thus c_k will report a table requirement of $2k$ to its parent, and $tabreq(G^k) \geq 2k$. ■

4 Memory requirement

Minimal separators of a partial k-tree can have varying sizes. The size of a minimal separator, plus 1, is a lower bound on the size of the corresponding bag of a tree-decomposition T, which in turn determines the size of the table associated with the bag. These tables of varying size are updated during the dynamic programming computations on T.

In this section, we consider the case of variable size tables stored at nodes of the tree-decomposition T. In this case, minimizing table requirement is not enough. Instead, we would like to minimize the memory requirement, *i.e.*, the sum of the sizes of tables that must be stored simultaneously, over all bottom-up traversal orders of T. An efficient algorithm solving this problem still eludes us and we leave its design as an open question. Instead, we consider only traversals of T that follow a depth-first search (dfs) traversal order.

Definition 4.1 *An ordering of edges $e_1, ..., e_{n-1}$ of a tree T on $n > 1$ vertices is a dfs traversal if we can choose a root r of T and for each vertex v of T_r order its children $child_1(v), ..., child_{c_v}(v)$ so that a call of the procedure DFS(r) of Section 2 will execute the instructions update-table(u, v) for edges $\{u, v\}$ in the order $e_1, ..., e_{n-1}$.*

For completeness, we also define a top-down traversal:

Definition 4.2 *An ordering of edges $e_1, ..., e_{n-1}$ of a rooted tree T_r on $n > 1$ vertices is a* top-down traversal *if for any edge $e_j = (u, parent(u))$ all edges in the subtree rooted in u have indices greater than j.*

Note that the traversal minimizing table requirement given in Section 2 is a dfs traversal. The implementation of a dfs traversal of a tree-decomposition is very simple, as illustrated by the DFS procedure given in Section 2. We will give a two-phase linear-time algorithm to find a dfs traversal order of a tree with weighted vertices minimizing its dfs memory requirement. Let $tab(u)$ be the size of the table at a node u of T. We define $mem(u, v)$ as the minimum dfs memory requirement of the subtree rooted in u with overall root chosen so that u has parent v, taken over all dfs traversals of this subtree.

Theorem 4.3 *In a given rooted tree, let a non-root vertex u have the parent v and let x and y be the children of u (if any) such that $mem(x, u) \geq mem(y, u) \geq$*

$mem(w,u)$ for any other child w of u (if u has only one child define $mem(y,u)$ to be 0.) Then, the minimum dfs memory requirement to compute the table associated with u is given by

$$mem(u,v) = \begin{cases} tab(u) & \text{for a leaf } u \\ \max\{mem(x,u), & \\ \quad tab(x) + tab(u), & \\ \quad mem(y,u) + tab(u)\} & \text{for a non-leaf } u. \end{cases} \tag{3}$$

The proof of this theorem is analogous to the proof of Theorem 2.2, in which case table sizes are uniformly 1. We therefore leave the proof out, except for noting that the necessity of the value $mem(y,u) + tab(u)$ follows since we are restricting to dfs traversals. We define the dfs memory requirement of T_r, $mem(T_r)$, as the minimum dfs memory requirement necessary to complete the table at r.

Corollary 4.4 *The right hand side of Theorem 3 can also be used to compute $mem(T_u)$; in this case, all neighbors of u are considered in defining x and y.*

For any vertex r of tree T with neighbors $w_1, \ldots, w_c$, the values of dfs memory requirements for the principal subtrees of T_r, $mem(w_1, r), \ldots, mem(w_c, r)$, together with $tab(r)$, determine the value of $mem(T_r)$. For a pair of adjacent vertices u, v, the value of $mem(u,v)$ does not depend on the particular chosen root, only that the root is in the component of $T \setminus (u,v)$ that includes v. We therefore conclude that knowledge of table sizes and the values $mem(u,v)$ and $mem(v,u)$, for each edge (u,v) of T, provides all the information necessary to compute $mem(T_r)$, for any vertex r of T. The following two-phase algorithm utilizes this observation to find an optimal root and traversal order.

Let x be an arbitrary vertex of T.

1. **Bottom-up** Pick a root x and using Theorem 3, compute in an arbitrary bottom-up traversal of T_x the values $mem(u,v)$ for all edges uv of T where v is the parent of u in T_x.

2. **Top-down** Using Theorem 3, compute in an arbitrary top-down traversal of T_x the values $mem(u,v)$ for all edges uv of T where v is the child of u in T_x. When processing a vertex u, compute also $mem(T_u)$ using the right-hand side of Theorem 3 with children of u ranging over all its neighbors. During the traversal, select a vertex r that has the minimum value of $mem(T_r)$.

The vertex r above is an optimal root. To find a traversal order minimizing dfs memory requirement let a node v in T_r have children $child_1(v), ..., child_{c_v}(v)$, with $child_1(v)$ having the largest value of $mem(child_i(v), v)$ over $1 \le i \le c_v$, as computed by the algorithm. These values are the dfs memory requirements for subtrees of T_r

rooted at these vertices. A call of DFS(r) of the procedure in section 2 will then perform the dynamic programming on T while minimizing the sum of the sizes of tables stored simultaneously, over all dfs traversals of the tree.

See Figure 3 for an example. At each node v of the tree we keep track of both the largest, second-largest and third-largest (these could be equal) dfs memory requirements $mem(w, v)$ reported by neighbors of v, and the identity of these neighbors. These values are incrementally updated. Assume that v has the neighbors r, x, y, z, w, with the bottom-up phase resulting in the values $mem(x, v) \geq mem(y, v) \geq mem(z, v) \geq mem(w, v)$. The three largest values are stored, and the two largest values are used in the bottom-up phase to compute $mem(v, r)$. In the top-down phase, the values used to compute, say, $mem(v, x)$ will be $mem(y, v)$ and the maximum of $mem(r, v)$ and $mem(z, v)$. These values and the identities of the corresponding neighbors are also used in the eventual call of DFS(r).

Theorem 4.5 *The algorithm given above computes an optimal root r minimizing dfs memory requirement in linear time, and the call of DFS(r) gives an optimal traversal order.*

Proof. The correctness of the bottom-up phase of the algorithm follows by Theorem 4.3. By Corollary 4.4, the computation of $mem(T_u)$ for a vertex u in the top-down phase yields the correct value of the minimum dfs memory requirement with that vertex as the root. Using the data structures mentioned above (with values of the three largest memory requirements for subtrees of each node incrementally updated) each visit in the traversals takes constant time, hence the linear time complexity. The call of DFS(r) results in a traversal order minimizing table requirement as described in the proof of Theorem 4.3 and Theorem 2.2. ∎

5 Conclusions and Future Research

Recent efforts have been aimed at making the theoretically efficient dynamic programming algorithms on bounded treewidth graphs amenable also to practical applications. Thorup [14] has shown that the control-flow graphs of goto-free programs have treewidth bounded by 6, and that an optimal tree-decomposition of the graph can be found easily. This result has been used to give a linear-time algorithm that takes a goto-free program and for a fixed number of registers determines if register allocation can be done without spilling [2]. A key ingredient in this latter algorithm is dynamic programming on a bounded treewidth graph.

We have identified and described the problem of table and memory requirements of table computations in dynamic programming algorithms on bounded treewidth graphs and we have provided linear-time algorithms that can be applied in a preprocessing stage to minimize the effects of this problem. In Section 1 we mentioned

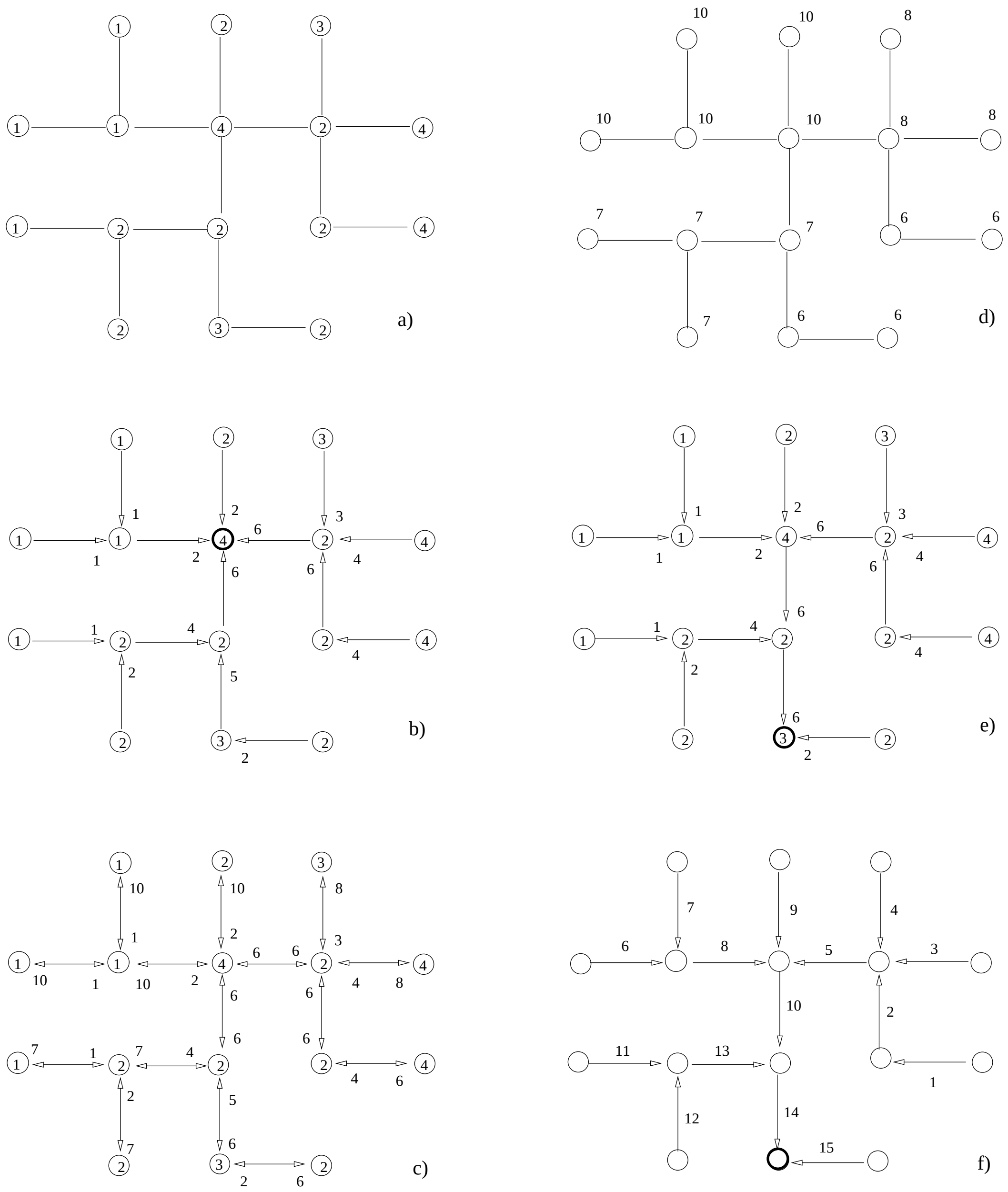

Figure 3: a) An unrooted tree with table sizes for each node. b) Bottom-up phase of algorithm with arbitrary root in bold and dfs memory requirements for subtrees. c) Top-down phase of algorithm. d) dfs memory requirements by rooting at respective nodes. e) Minimum dfs memory requirement with optimal root in bold. f) An optimal dfs traversal order with root in bold.

the example of a width 10 tree-decomposition of 1000 nodes, with uniform table-size of 8 Kbytes. By applying Corollary 2.4 we see that the traversal order given by the algorithm of Section 2 for this example uses at most 10 tables, giving a memory requirement of 80 Kbytes.

Using our algorithms in a pre-processing stage it is possible to apply the dynamic programming stage of the partial k-tree algorithms to much larger input graphs, or graphs with larger treewidth, without paying the cost of external memory references. The overhead associated with this linear-time, small-constant pre-processing, is in most cases negligible in comparison with the cost of the dynamic programming itself, which for any NP-hard problem on an n-node treewidth k graph has a running time of $\Omega(nc^k)$ for some constant c (unless there are sub-exponential time algorithms for NP-hard problems). However, more programming work is needed to confirm this experimentally.

From Theorem 2.2 it is clear that table requirement two is achieved when there is a width k tree-decomposition T of the input graph G where T is a caterpillar, *i.e.* T is a tree obtained from a path by adding leaves adjacent to the path, or equivalently T is a connected graph of pathwidth 1. When T is a path, the treewidth and pathwidth of G coincide. This observation thus extends the "folklore" that dynamic programming on a path-decomposition (a path) is simpler than on a general tree-decomposition. A natural question to ask is for a characterization of graphs of treewidth k that have a width k tree-decomposition forming a caterpillar. A partial answer to this question is presented in [10]: every multitolerance graph has an optimal tree-decomposition forming a caterpillar. The more general question is how to find tree-decompositions having small value of memory requirement.

Finally, the design of an efficient algorithm finding the minimum memory requirement of a tree with varying table size, over all bottom-up traversal orders, remains open. Another interesting question is that of bounding the worst case performance ratio of the DFS-traversal algorithm over this minimum value.

Acknowledgements We thank the two anonymous referees for many valuable suggestions that have improved both the contents and the presentation of the results.

References

[1] T. Beyer, S.M. Hedetniemi, S.T. Hedetniemi and A. Proskurowski, Graph traversal with minimum stack depth, *Congressus Numerantium, Vol. 32* , 121-130, 1981.

[2] H. Bodlaender, J. Gustedt and J.A. Telle, Linear-time register allocation for a fixed number of registers and no stack variables, *Proceedings 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, 574-583.

[3] H. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, in *Proceedings 25th Symposium on the Theory of Computing* (STOC'93), 226-234.

[4] H. Bodlaender, A tourist guide through treewidth, *Acta Cybernetica,* 11:1-21, 1993.

[5] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff and J.S.Vitter, External-Memory Graph Algorithms, *Proc. ACM-SIAM Symp. on Discrete Algorithms* (SODA'95), pp. 139–149, 1995.

[6] J.A. Ellis, I.H. Sudborough and J.S. Turner, The vertex separation number and search number of a graph, *Information and Computation* vol. 113, 50-79, 1994.

[7] B. Hiim, Implementing and testing algorithms for tree-like graphs, Master's Thesis, November 1997, Dept. of Informatics, University of Bergen.

[8] L. Kirousis and C. Papadimitriou, Searching and pebbling, *Theoretical Computer Science* 47, 205-218, 1986.

[9] R. Möhring, Graph problems related to gate matrix layout and PLA folding, in *Computational Graph Theory, Computing Suppl. 7*, Springer-Verlag, 17-51, 1990.

[10] A.Parra, Triangulating multitolerance graphs, Technical Report 392/1994, TU Berlin, Germany, 1994.

[11] N. Robertson and P. Seymour, Graph Minors I. Excluding a forest, *Journal of Combinatorial Theory Series B* 35, 39-61, 1983.

[12] P. Scheffler, A linear algorithm for the pathwidth of trees, *Topics in Combinatorics and Graph Theory, Physica-Verlag Heidelberg*, 613-620, 1990.

[13] J.A. Telle and A. Proskurowski, Algorithms for vertex partitioning problems on partial k-trees, *SIAM Journal on Discrete Mathematics*, Vol. 10, No. 4, 529-550, November 1997.

[14] M. Thorup, Structured Programs have Small Tree-Width and Good Register Allocation, *Proceedings 23rd Workshop on Graph-Theoretical Concepts in Computer Science (WG'97)*, LNCS vol. 1335, 318-332.