REPORTS
IN
INFORMATICS

ISSN 0333-3590

Dynamic programming on tree-decompositions versus branch-decompositions

Frederic Dorn and Jan Arne Telle

REPORT NO 293 **April 2005**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL `http://www.ii.uib.no/publikasjoner/texrap/ps/2005-293.ps`

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at `http://www.ii.uib.no/publikasjoner/texrap/`.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Dynamic programming on tree-decompositions versus branch-decompositions

Frederic Dorn and Jan Arne Telle

Department of Informatics, University of Bergen,
Bergen, Norway

Abstract We introduce semi-nice tree-decompositions that combine the best properties of tree-decompositions, branch-decompositions and path-decompositions. We give dynamic programming algorithms on semi-nice tree-decompositions for various optimization problems and show how the so-called D,E,F-partition of its Join nodes allows for a more refined runtime analysis. We advocate a two-step approach of first transforming a given branch-decomposition, tree-decomposition or path-decomposition into a semi-nice tree-decomposition and then performing dynamic programming on this latter structure. We employ this approach to problems that in the litterature have been solved using dynamic programming directly on a given branch-decomposition or tree-decomposition and in each case we match or improve their running time.

1 Introduction

The three graph parameters treewidth, branchwidth and pathwidth were all introduced by Robertson and Seymour as tools in their seminal proof of the Graph Minors Theorem. The treewidth $tw(G)$ and branchwidth $bw(G)$ of a graph G satisfy the relation $bw(G) \leq tw(G) + 1 \leq \frac{3}{2}bw(G)$ [10], and thus whenever one of these parameters is bounded by some fixed constant on a class of graphs, then so is the other. Tree-decompositions have traditionally been the choice when solving NP-hard graph problems by dynamic programming to give FPT algorithms when parameterized by treewidth, see e.g. [3] for an overview. Of the various algorithmic templates suggested for this over the years the nice tree-decompositions [8] with binary Join and unary Introduce and Forget operations are preferred for their simplicity and have been widely used both for showing new results, for pedagogical purposes, and in implementations. Recently there have been some papers [7,5,4] whose results suggest that the base of the exponent in the running time of these FPT algorithms could be improved if one instead uses branch-decompositions.

In this paper we argue for the opposite view, namely that dynamic programming on branch-decompositions is no faster than on tree-decompositions. For this purpose we introduce semi-nice tree-decompositions that maintain much of the simplicity of the nice tree-decompositions but with a binary Join operation that allows D-vertices that appear in only one of the children bags. During dynamic programming these D-vertices, and also F-vertices for which all neighbors have already been considered, are treated specially in the Join update operation. Depending on the number of D-vertices and F-vertices this will improve the running time for the Join update operation. Along the way we also simplify the proof of monotonicity of table entries for domination-type problems of [1] by a slight change in the definition of the vertex states used. Our results are also a step towards meeting the 'research challenge', first proposed by Alber and Niedermeier in [2], of lowering to $O(n\lambda^k)$ the runtime of dynamic programming on treewidth k graphs for solving a problem having λ vertex-states.

To compete with the recent results using branch-decompositions, we show how to first transform a given branch-decomposition into a semi-nice tree-decomposition and then do dynamic programming on this latter structure. We employ this approach to various problems, such as minimum dominating set solved in [7] and (k, r)-center solved in [5] in each case maintaining the running time of the algorithms on branch-decompositions given in those papers. For graphs having treewidth close to branchwidth we show that the fastest algorithms are actually achieved by transforming an optimal tree-decomposition, rather than branch-decomposition, into a semi-nice tree-decomposition and do dynamic programming on this latter structure. In this way the semi-nice tree-decompositions combine, by a single dynamic programming algorithm, the best performances achievable either on tree-decompositions or branch-decompositions, and in fact also path-decompositions.

2 Definitions

We use standard graph notation and terminology, e.g. for a subset $S \subseteq V(G)$ of the vertices of a graph G we denote by $N(S)$ the set of vertices not in S that are adjacent to some vertex in S. For clarity we speak of nodes of a tree and vertices of a graph. To simplify expressions involving the cardinality of a set X, we write e.g. 2^X when we actually mean $2^{|X|}$.

A tree-decomposition $(T, \mathcal{X})$ of a graph G is an arrangement of the vertex subsets $\mathcal{X}$ of G, called bags, as nodes of the tree T such that for any two adjacent vertices in G there is some bag containing them both, and for each vertex of G the bags containing it induce a connected subtree. When we say bag we may refer both to the tree node and the associated vertex subset, sometimes even both at the same time, e.g. 'the intersection of two adjacent bags'. The width of the tree-decomposition $(T, \mathcal{X})$ is simply the size of the largest bag minus one.

A branch-decomposition (T, μ) of a graph G is a ternary tree T, i.e. with all inner nodes of degree three, together with a bijection μ from the edge-set of G to the leaf-set of T. For every edge e of T define a vertex subset of G called $mid(e)$ consisting of those vertices $v \in V(G)$ for which e lies on the path in T between two leaves whose mapped edges are incident to v (note that this is a non-standard but equivalent way of defining these so-called middle sets.) The width of (T, μ) is the size of the largest $mid(e)$ thus defined.

For a graph G its treewidth and branchwidth is the smallest width of any tree-decomposition and branch-decomposition of G, respectively, while its pathwidth is the smallest width of a tree-decomposition $(T, \mathcal{X})$ where T is a path.

3 Semi-nice tree-decompositions

In this section we introduce semi-nice tree-decompositions and show how to do dynamic programming on them, using domination-type problems as our case-study. A tree-decomposition $(T, \mathcal{X})$ is *semi-nice* if T is a rooted binary tree with each non-leaf of T being either a:

- **Introduce** node X with a single child C and $C \subset X$.
- **Forget** node X with a single child C and $X \subset C$.
- **Join** node X with two children B, C and $X = B \cup C$.

For an Introduce node we call $X \setminus C$ the 'introduced vertices' and for a Forget node $C \setminus X$ the 'forgotten vertices'. It follows by properties of a tree-decomposition that a vertex can be introduced in several nodes but is forgotten in at most one node. Note that the nice tree-decompositions [8] require that a Join node has $X = B = C$, Introduce has $|X| = |C|+1$, and Forget has $|X| = |C|-1$, but are otherwise identical to the semi-nice tree-decompositions.

For a Join node X with children B, C and parent A (the root node being its own parent) we define a partition of $X = B \cup C$ into 3 sets D, E, F:

- **Symmetric Difference** $D = (C \setminus B) \cup (B \setminus C)$
- **Expensive** $E = A \cap B \cap C$
- **Forgettable** $F = (B \cap C) \setminus A$

The *Forgettable* vertices are useful whenever a node X has a Forget parent A, and their definition for a non-Join node X with parent A is simply $F = X \setminus A$. We say that a neighbor u of a vertex $v \in X$ has been *considered* at node X of T if $u \in X$ or if $u \in X'$ for some descendant node X' of X. Clearly, if X is a Forget node forgetting v then all neighbors of v must have been considered at X. For fast dynamic programming we want sparse semi-nice tree-decompositions where vertices are forgotten as soon as possible.

Definition 1. *In a* sparse *semi-nice tree-decomposition, if a node X has the property that all neighbors of a vertex $v \in X$ have been considered then the parent of X is a Forget node forgetting v.*

In particular this implies that in a Join node with parent A and children B, C any vertex in $B \setminus A \cup C$ has a neighbor in $C \setminus A \cup B$ and vice-versa. Otherwise such a vertex could be forgotten by a child of the Join node. We call such edges *new edges* since these adjacencies have not been considered in any of the child nodes.

Given a tree-decomposition $(T, \mathcal{X})$ of width k of a graph G with n vertices, we can in time $O(k^2 n)$ make it into a sparse semi-nice tree-decomposition $(T', \mathcal{X}')$ of width k while keeping the E-sets in the partition of each Join node as small as the given tree-decomposition allows. For space reasons we give only a sketch of the algorithm. Choose a root and transform T into a binary tree as follows. For each node X having parent A and $d \geq 3$ children $C_1, C_2, ..., C_d$ replace X by a path of $d-1$ nodes, each one with bag X, with one end-node of the path being a child of A, another end-node of the path having child C_d and the remaining $d-1$ children distributed one to each node on the path. Process the tree bottom-up to find for each vertex $v \in V(G)$ a lowest node X_v at which all neighbors of v have been considered, and remove v from any bag that is not a descendant of X_v. Then, for each node X with a single child C such that both $X \setminus C$ and $C \setminus X$ are nonempty, subdivide the edge between X and C by a Forget node with bag $C \cap X$. Then, for each node X with two children B, C if $X \setminus (B \cup C)$ is nonempty make a new parent bag $B \cup C$ for B and C and make X the parent of this new bag. Finally, for each node X with two children B, C if $B \setminus X$ is nonempty then subdivide the edge between X and B by a Forget node with bag $X \cap B$, and likewise for C. The result is a sparse semi-nice tree-decomposition.

We now explain the algorithmic template for doing fast dynamic programming on a semi-nice tree-decomposition $(T, \mathcal{X})$ of a graph G to solve an optimization problem on G . As usual, we compute in a bottom-up manner along the rooted tree T a table of solutions for each node X of T. Let G_X denote the subgraph of G induced by vertices $\{v \in X' : \mathrm{X}' = X \text{ or } X' \text{ a descendant of } X \text{ in } T\}$. The table $Table_X$ at X will store solutions to the optimization problem on G_X indexed by certain equivalences classes of solutions. The solution to the problem on G is found by an optimization over the table at the root of T. To develop a specific algorithm one must define the tables involved and then show how to Initialize the table at a leaf node of T, how to compute the tables of Introduce, Forget and Join nodes given that their children tables are already computed, and finally how to do the Optimization at the root.

We use the Minimum Dominating Set problem as an example, whose tables are described by the use of three so-called vertex states:

- **Dom** (Dominating)
- **NbrD** (Neighbor is Dominating)
- **Free** (Temporary state)

Each index s of $Table_X$ at a node X represents an assignment of states to vertices in the bag X. For index $s : X \rightarrow \{Dom, NbrD, Free\}$ the vertex subset S of G_X is *legal* for s if:

- $V(G_X) \setminus X = (S \cup N(S)) \setminus X$
- $\{v \in X : s(v) = Dom\} = X \cap S$
- $\{v \in X : s(v) = NbrD\} \subseteq X \cap N(S)$

$Table_X(s)$ is defined as the cardinality of the smallest S legal for s, or $Table_X(s) = \infty$ if no S is legal for s.

Informally, the 3 constraints are that S is a dominating set of $G_X \setminus X$, that vertices with state Dom are exactly $X \cap S$, and that vertices with state NbrD have a neighbor in S. Note that vertices with state Free are simply constrained not to be in S. Since this is also a constraint on vertices with state NbrD a subset S which is legal for an index s would still be legal even if some vertex with state NbrD instead had state Free. This immediately implies the monotonicity property $Table_X(t) \leq Table_X(s)$ for pairs of indices t and s where $\forall v \in X$ either $t(v) = s(v)$ or $t(v) = Free$ and $s(v) = NbrD$.

Let us also remark that the $Table_X$ data structure should be an array. To simplify the update operations we should associate integers 0,1,2 with each vertex state so that an index is a 3-ary string of length $|X|$. Moreover, the ordering of vertices in the indices of $Table_X$ should respect the ordering in $Table_C$ for any child node C of X and in case C is the only child of X then all vertices in the larger bag should precede those in the smaller bag. We find this by computing a total order on $V(G)$ respecting the partial order given by the ancestor/descendant relationship of the Forget nodes forgetting vertices $v \in V(G)$.

The table $Table_X$ at a Forget node X will have 3^X indices, one for each of the possible assignments $s : X \rightarrow \{Dom, NbrD, Free\}$. We assume a machine model with words of length 3^X, to avoid complexity issues related to fast array accesses. Assume Forget node X has child C with $Table_C$ already computed. The correct value for $Table_X(s)$ is the minimum of $\{Table_C(s^+)\}$ over all indices s^+ where $s^+(v) = s(v)$ if $v \in X$ and $s^+(v) \in \{Dom, NbrD\}$ otherwise. For this reason we call the state Free a Temporary state. The Forget update operation takes time $O(3^X 2^{C \setminus X})$.

Note that the Forget update operation had no need for the indices of the table at the child where a forgotten vertex in $C \setminus X$ had state Free. This observation allows us to save some space and time for the Forgettable vertices of a bag having a Forget parent.

If X is a leaf node with Forgettable vertices F then $Table_X$ has only $3^{X \setminus F} 2^F$ indices, in accordance with the above observation, and is computed in a brute-force manner. This takes time $O(X 3^{X \setminus F} 2^F)$, since for each index s we must check if $Table_X(s)$ should be equal to the number of vertices in state Dom, or if there is a vertex in state NbrD with no neighbor in state Dom in which case $Table_X(s) = \infty$.

If X is an Introduce node with Forgettable vertices F and child C then $Table_X$ has $3^{X \setminus F} 2^F$ indices and the correct value at $Table_X(s)$ is:

- ∞ if $Table_C(s) = \infty$ or if $\exists x \in X \setminus C$ with $s(x) = NbrD$ but no neighbor of x in state Dom.
- $Table_C(s) + |\{v \in X \setminus C : s(v) = Dom\}|$ otherwise

The Introduce update operation thus takes time $O(X3^{X\setminus F}2^F)$.

The correct values for $Table_X$ at a Join node X with partition D, E, F and children B, C are computed in three steps, where the last two steps account for new adjacencies that have not been considered in any child table (we call these 'new edges'):

1. $\forall s : Table_X(s) = \min\{Table_B(s_b) + Table_C(s_c) - |B \cap C \cap \{v : s(v) = Dom\}|\}$ over (s_b, s_c) such that triple (s, s_b, s_c) is necessary (see below).
2. $\forall R \subseteq D : New(R) = (N(C \cap R) \cap (D \cap B \setminus R)) \cup (N(B \cap R) \cap (D \cap C \setminus R))$
3. $\forall s : Table_X(s) = Table_X(s')$ where $s'(v) = Free$ if $v \in D \wedge s(v) = NbrD \wedge v \in New(\{u : s(u) = Dom\})$ and otherwise $s'(v) = s(v)$.

We describe and count the necessary triples of indices (s, s_b, s_c) for the Join update using the method of [7], by first considering the number of necessary vertex state triples $(s(v), s_b(v), s_c(v))$ such that vertex state $s_b(v)$ and $s_c(v)$ in B and C respectively will yield the vertex state $s(v)$ in X:

- $v \in B \setminus C \subseteq D$: 3 triples (Dom,Dom,-), (NbrD,NbrD,-), (Free,Free,-)
- $v \in C \setminus B \subseteq D$: 3 triples (Dom,-,Dom), (NbrD,-,NbrD), (Free,-,Free)
- $v \in F$: 3 triples (Dom,Dom,Dom), (NbrD,Free,NbrD), (NbrD,NbrD,Free)
- $v \in E$: 4 triples (Dom,Dom,Dom), (NbrD,Free,NbrD), (NbrD,NbrD,Free), (Free,Free,Free)

Lemma 1. *The Join update just described for a node X with partition D, E, F is correct and takes time $O(3^{D+F}4^E)$.*

The proof of this lemma is for space reasons moved to the appendix. Finally, at the root node R of T we compute the smallest dominating set of G by the minimum of $\{Table_R(s) : s(v) \in \{Dom, NbrD\} \forall v \in R\}$. This takes time $O(2^R)$.

Correctness of the algorithm follows by induction on the tree-decomposition, in the standard way for such dynamic programming algorithms.

For the timing we have the Join operation usually being the most expensive, although there are graphs, e.g. when pathwidth=treewidth, for which the leaf Initialization or Introduce operations are the most expensive. However, the Forget and Root optimization operations will never be the most expensive.

Theorem 1. *Given a semi-nice tree-decomposition $(T, \mathcal{X})$ of a graph G on n vertices we can solve the Min Dominating Set Problem on G in time $O(n(\max\{4^E3^{D+F}\} + \max\{X3^{X\setminus F}2^F\}))$ with maximization over Join nodes of T with partition D, E, F and over Initialization and Introduce nodes with bag X and Forgettable set F, respectively.*

3.1 Other Domination-type problems

In this section we extend our result from the previous section to problems over vertex subsets having other domination-type constraints. A general class of such constraints are parameterized by two subsets of natural numbers σ and ρ. A subset of vertices S is a (σ, ρ)-set if $\forall v \in S$ we have $|N(v) \cap S| \in \sigma$ and $\forall v \notin S$ we have $|N(v) \cap S| \in \rho$ [11]. Some well-studied and natural types of (σ, ρ)-sets are when σ is either all natural numbers, all positive numbers or $\{0\}$, and ρ is either all positive numbers or $\{1\}$. The six resulting constraints are called Dominating set ($\sigma = nat$, $\rho = pos$); Perfect Dominating Set ($\sigma = nat$,$\rho = \{1\}$); Independent Dominating set ($\sigma = \{0\}$,$\rho = pos$); Perfect Code ($\sigma = \{0\}$,$\rho = \{1\}$); Total Dominating set ($\sigma = pos$, $\rho = pos$); Total Perfect Dominating set ($\sigma = pos$,$\rho = \{1\}$). For Perfect Code and Total Perfect Dom set it is NP-complete simply to decide if a graph has any such set, for Ind Dom set it is NP-complete to find either a smallest or largest such set, while for the remaining three problems it is NP-complete to find a smallest set. The paper [2] considers these six constraints, and give dynamic programming algorithms on nice tree-decompositions that take into account monotonicity properties to arrive at fast runtimes. According to our taste the treatment of some of these problems in [2] is too short to enlighten the subtle but crucial differences among the problems. We give a slightly more detailed description of the algorithms, while also improving the runtime by considering semi-nice tree-decomposition.

We start by Perfect code, which is not an optimization problem, since any Perfect Code in a graph has the same size. However, it is NP-complete to decide if an arbitrary graph has a Perfect Code [9]. We construct an algorithm using semi-nice tree-decompositions, using the same terminology and variables as in the Dominating set algorithm.

We use the 3 vertex states **Dom**, **1NbrD** and **0NbrD**. For index $s : X \to \{Dom, 0NbrD, 1NbrD\}$ the vertex subset S of G_X is *legal* for s if: S is a perfect code of $G_X \setminus X$ (i.e. every vertex $v \in V(G_X) \setminus X$ has $|N[v] \cap S| = 1$), vertices with state Dom are exactly $X \cap S$ and they have no neighbors with state Dom, vertices with state 1NbrD have exactly one neighbor in $S \setminus X$, and vertices with state 0NbrD have zero neighbors in $S \setminus X$. Note that 0NbrD and 1NbrD reflect only the number of dominating neighbors *outside* of the bag X. We sketch the further differences with the Dominating set algorithm.

A table index s stores True if there exists any vertex subset legal for s, and False otherwise. During Forget update we consider only indices in the child table $Table_C$ where the forgotten vertices are either in state Dom, or in state 1NbrD and none of its neighbors in C are in state Dom, or in state 0NbrD and exactly one of its neighbors in C are in state Dom. During Root-optimization at X we look for an index storing True having the property that any vertex with state 1NbrD has no neighbors in X with state Dom, and any vertex with state 0NbrD has exactly one neighbor in X with state Dom.

During the Join update the following 4 triples are necessary for vertices E and F: (Dom,Dom,Dom), (0NbrD,0NbrD,0NbrD), (1NbrD,0NbrD,1NbrD), (1NbrD,1NbrD,0NbrD). For the dominating set problem the triple (NbrD,NbrD,NbrD) was not necessary because of a monotonicity property between the NbrD and Free states. The argument that we don't need the triple (1NbrD,1NbrD,1NbrD) is that 1NbrD reflects a dominating neighbor outside the bag X and Perfect Code asks for only one dominating neighbor. For the vertices in D we have 3 necessary triples, one for each vertex state. $Table_X(s)$ at a Join node X with children B, C is the disjunction of conjunctions $Table_B(s_b) \wedge Table_C(s_c)$ over all pairs of indices (s_b, s_c) such that the triple (s, s_b, s_c) is a necessary triple of indices. Then, set $Table_X(s) = False$ if for some new edge uv $s(u) = s(v) = Dom$. The timing for the Join for the Perfect Code problem is thus $O(3^D 4^{E+F})$, by a proof similar to the one for Lemma 1.

We now turn to Total Dominating set where the vertices in the dominating set S also must have at least one neighbor in S. We use the 4 vertex states **DomNbrD**, **NbrD**, **DomFree**, and **Free**, where the two latter are temporary states. For index $s : X \to \{DomNbrD, NbrD, DomFree, Free\}$ the vertex subset S of G_X is *legal* for s if: S is a total dominating set of $G_X \setminus X$, vertices with state DomNbrD or DomFree are exactly $X \cap S$, and vertices with state DomNbrD or NbrD have a neighbor in S. Note that vertices with state DomFree and Free are simply constrained to be in S or not in S, respectively. Since these are also constraints on vertices with state DomNbrD and NbrD, respectively, we have the monotonicity property $Table_X(t) \leq Table_X(s)$ for pairs of indices t and s where $\forall v \in X$ either $t(v) = s(v)$ or $t(v) = DomFree$ and $s(v) = DomNbrD$ or $t(v) = Free$ and $s(v) = NbrD$.

The algorithm is very similar to the Dominating set algorithm, except that also vertices in the dominating set have temporary state DomFree in addition to state DomNbrD. During the Join update 6 triples are necessary for Expensive vertices: (Free,Free,Free), (NbrD,NbrD,Free),(NbrD,Free,NbrD), (DomFree,DomFree,DomFree), (DomNbrD,DomNbrD,DomFree), (DomNbrD,DomFree,DomNbrD). The first three triples occur also in the dominating set algorithm, while the argument that the three last triples for dominating vertices suffice is that the monotonicity property holds also between the DomNbrD and DomFree states. Thus 6 triples total and runtime $O(6^k)$ for a Join update if all vertices are Expensive. Note that [2] uses a different counting argument and claims runtime $O(5^k)$, but this is because they have forgotten a factor $2^{n_i-z_0-z_1}$ in their products of binomial coefficients formula, and adding this factor they would also arrive at $O(6^k)$. For the vertices in D we get 4 triples simply because we have 4 vertex states, while for vertices in F we get the 4 triples (NbrD,NbrD,Free),(NbrD,Free,NbrD) (DomNbrD,DomNbrD,DomFree), (DomNbrD,DomFree,DomNbrD) since Free and DomFree are temporary states. The timing for the Join for the Min Total Dominating set problem is thus $O(4^{D+F}6^E)$.

For Independent Dominating Set we get the same runtime as Dominating Set, while for Perfect Dominating Set we get the same runtime as Perfect Code. For Total Perfect Dominating set we combine our solutions for Perfect Dominating set and Total Dominating set for a runtime for Join of $O(4^D 5^F 6^E)$. Let us end by noting that these results can be generalized to solve optimization problems over general (σ, ρ)-sets. For example, if $\sigma = \{0, 1, ..., p\}$ and $\rho = \{0, 1, ..., q\}$ we are asking for a subset $S \subseteq V(G)$ such that S induces a subgraph of maximum degree at most p with each vertex in S having at most q neighbors in $V(G) \setminus S$. For this case we would use $p+q+2$ vertex states and get runtime $O((p+q+2)^D(s(p)+s(q))^{E+F})$, where $s(i)$ is the number of pairs of ordered non-negative integers summing to i. Thus, for the Maximum 2-Packing problem (also known as Max Strong Stable set), which is of this form with $p = 0$ and $q = 1$, we get an $O(3^D 4^{E+F})$ algorithm. For the more general cases we believe there is room for improvement. The previous best results for these problems [2] correspond to our results when treating all vertices as Expensive, so we have moved closer to the goal of λ^{D+E+F} time for a problem with λ vertex states. However, the worst-case runtime for general graphs remains unbeaten since there are graphs, for example k-trees, where all vertices in a bag are Expensive. In the next section we show that the runtime of these algorithms can also be expressed as a function of the branchwidth of the graph. The results for domination-type problems are summarized in Table 4.

4 Runtime by branchwidth, treewidth or pathwidth

In this section we show that given an optimal branch-decomposition we can transform it into a semi-nice tree-decomposition and express the runtime of dynamic programming algorithms on this latter structure as a function of branchwidth bw, and similarly as a function of treewidth tw and pathwidth pw. The runtime expressed by bw matches the best results achieved by dynamic programming directly on the branch-decomposition. See Figure 6 in the appendix for an illustration of the transformation described in the following lemma.

Lemma 2. *Given a branch-decomposition (T, μ) of a graph G with n vertices and m edges we can in time $O(m)$ compute a sparse semi-nice tree-decomposition $(T', \mathcal{X})$ with $O(n)$ nodes such that for any bag X of T' we have some $t \in V(T)$ with incident edges e, f, g such that $X \subseteq mid(e) \cup mid(f) \cup mid(g)$ and if X is a Join node with partition D, E, F then $E \subseteq mid(e) \cap mid(f) \cap mid(g)$ and $F \subseteq mid(f) \cap mid(g) \setminus mid(e)$ and $D \subseteq mid(e) \setminus mid(f) \cap mid(g)$.*

Proof: The algorithm has 3 steps: 1) Transform branch-decomposition into a tree-decomposition on the same tree, 2) Transform tree-decomposition into a small tree-decomposition having $O(n)$ nodes, 3) Transform small tree-decomposition into a sparse semi-nice tree-decomposition. Step 1) is well-known (see e.g. [7] for a correctness proof) and proceeds as follows: an inner node t with incident edges e, f, g gets bag $X_t = mid(e) \cup mid(f) \cup mid(g)$, and a leaf node t gets bag containing the two adjacent vertices making the edge $\mu^{-1}(t)$. Root the tree arbitrarily in a leaf. Assume inner node X_t has parent A and children B, C on incident edges e, f, g, respectively. Note that by construction $X_t = mid(e) \cup mid(f) \cup mid(g)$, and $X_t \cap A = mid(e)$, $X_t \cap B = mid(f)$ and $X_t \cap C = mid(g)$. Assume X_t ends up like this as a Join node after step 3). The partition D, E, F for X_t is then by definition $E = A \cap B \cap C = mid(e) \cap mid(f) \cap mid(g)$, and $F = B \cap C \setminus A = mid(f) \cap mid(g) \setminus mid(e)$, which implies also $D = mid(e) \setminus mid(f) \cap mid(g)$, in agreement with the statement in the lemma. In step 2) we iteratively contract any edge between two nodes with at least one node of degree at most 2 whose bags X, C satisfy $C \subseteq X$ and leave the bag X on the contracted node. In step 3) we apply the algorithm from Section 3 that transforms a tree-decomposition into a sparse semi-nice tree-decomposition $(T', \mathcal{X})$. Steps 2) and 3) did not destroy the property that held for all inner nodes after step 1), and for every node X of T' we can find an original node $t \in V(T)$ with $X \subseteq X_t$. □

Corollary 1. *We can solve Minimum Dominating set by dynamic programming on a semi-nice tree-decomposition in time:* $O(2^{3\log_4 3bw}n) = O(2^{2.38\,bw}n)$ *if given a branch-decomposition* (T, μ) *of width bw;* $O(2^{2\,tw}n)$ *if given a tree-decomposition of width tw;* $O(2^{1.58\,pw}n)$ *if given a path-decomposition of width pw; and thus* $O(2^{\min\{1.58\,pw, 2\,tw, 2.38\,bw\}})$ *if given all three. For other domination-type problems we get the runtimes listed in Table 4.*

Proof: The first algorithm transforms (T, μ) into a sparse semi-nice tree-decomposition $(T', \mathcal{X})$ as in Lemma 2 and then applies the algorithm of Theorem 1. Consider a Join node X with partition D, E, F. By Lemma 2 D, E, F is related to an inner node $t \in V(T)$ with incident edges e, f, g by $E \subseteq mid(e) \cap mid(f) \cap mid(g)$ and $F \subseteq mid(f) \cap mid(g) \setminus mid(e)$ and $D \subseteq mid(e) \setminus mid(f) \cap mid(g)$. From our definition of middle sets it is easy to see that a vertex $v \in mid(e) \cup mid(f) \cup mid(g)$ appears in at least two out of $mid(e)$, $mid(f)$ and $mid(g)$. From this follows the constraint $|mid(e) \cup mid(f) \cup mid(g)| \leq 1.5\,bw$ which in addition to the constraints $|mid(e)| \leq bw, |mid(f)| \leq bw, |mid(g)| \leq bw$ gives us four constraints altogether. The worst-case runtime of the Join update of Theorem 1 is found by taking these four constraints as the constraints of a linear program maximizing $3^{D+F}4^E \leq 3^{|(mid(e)\cup mid(f)\cup mid(g))\setminus(mid(e)\cap mid(f)\cap mid(g))|}4^{|mid(e)\cap mid(f)\cap mid(g)|}$. The solution is computed by using an ordinary LP-solver and turns out to occur when $E = \emptyset$ and $|D + F| = 1.5bw$, which corresponds to $3^{1.5bw} = 2^{3\log_4 3bw}$. Note that for Introduce nodes we get the same worst-case bound.

In a tree-decomposition of width tw the worst-case occurs for a Join node with $tw + 1$ Expensive vertices, and Theorem 1 then gives runtime $O(2^{2\,tw}n)$. After transforming a path-decomposition of width pw, which of course is also a tree-decomposition, into a semi-nice tree-decomposition by the algorithm of Section 3, the latter will not have any Join nodes. The worst-case runtime occurs for a node X with $pw + 1$ vertices and empty Forgettable set, and Theorem 1 then gives runtime $O(pw\, 3^{pw}n) = O(2^{1.58\,pw}n)$. For the other domination-type problems we use an analogous argument. □

For certain classes of graphs, e.g. grid graphs, pathwidth is indeed the best parameter. The runtime we get for Minimum Dominating set as a function of branchwidth bw is essentially the same as that achieved by the algorithm of [7] working directly on the branch-decomposition (the runtime there is expressed with multiplicative factor m instead of our n but for a graph with branchwidth bw we have $m = O(n\,bw)$.) See Table 4 for a summary of the results for each domination-type

	States	Time for Join	Total time	Cutoff point
Min Dom set	3	$O(3^{D+F}4^E)$	$O(n2^{\min\{2tw,2.38bw\}})$	$tw \leq 1.19bw$
Min/Max Ind Dom set	3	$O(3^{D+F}4^E)$	$O(n2^{\min\{2tw,2.38bw\}})$	$tw \leq 1.19bw$
$\exists$ Perfect Code	3	$O(3^D4^{E+F})$	$O(n2^{\min\{2tw,2.58bw\}})$	$tw \leq 1.29bw$
Min Perfect Dom set	3	$O(3^D4^{E+F})$	$O(n2^{\min\{2tw,2.58bw\}})$	$tw \leq 1.29bw$
Max 2-Packing	3	$O(3^D4^{E+F})$	$O(n2^{\min\{2tw,2.58bw\}})$	$tw \leq 1.29bw$
Min Total Dom set	4	$O(4^{D+F}6^E)$	$O(n2^{\min\{2.58tw,3bw\}})$	$tw \leq 1.16bw$
$\exists$ Perfect Total Dom set	4	$O(4^D5^F6^E)$	$O(n2^{\min\{2.58tw,3.16bw\}})$	$tw \leq 1.22bw$

Table 1. The number of vertex states and time for a Join operation with Expensive vertices E, Forgettable vertices F and Symmetric Difference vertices D. Worst-case runtime expressed also by treewidth tw and branchwidth bw of the input graph, and the cutoff point at which treewidth is the better choice. To not clutter the table, we leave out pathwidth pw.

problem, expressed by tw and bw only, to not clutter the table, even though for each problem in the table such a cutoff point could be computed for which pw is best.

Let us end this section by remarking that the main difference between dynamic programming algorithms on a semi-nice tree-decomposition and on a branch-decomposition is that the subgraph of G for which solutions are stored in a table are in one case induced by a set of vertices and in the other case induced by a set of edges (additionally, we personally find it much easier to work with the terminology and structure of tree-decompositions, but this is a matter of taste.) Thus, to design a dynamic programming algorithm on semi-nice tree-decompositions for the (k, r)-center problem we can use the algorithm for branch-decomposition given in [5] as a basis. The main thing we must add is a procedure to handle the new edges among vertices in D, corresponding to steps 2 and 3 of our Join update procedure for Min Dom set. This is done in a way similar to our Min Dom set algorithm and the resulting algorithm for (k, r)-center on semi-nice tree-decompositions has the same runtime as a function of branchwidth as that given in [5]. For space reasons the sketch of the algorithm is put in the appendix. A comparison with the algorithm of [4] for the TSP problem is difficult to make as an exact runtime analysis is never given in that paper.

5 Future research

It is our hope that future research will improve further on the runtime analysis of dynamic programming on semi-nice tree-decompositions. For example, in a Join node X with parent A and partition D, E, F the vertices in $Z = D \setminus A$ will be forgotten in the parent node A. This raises the possibility of a Join update with faster runtime for these vertices (and a focus on Z, D, E, F-partitions for Join nodes.) Another issue is to get a better understanding of when pathwidth, treewidth or branchwidth is best, possibly relating this also to other graph parameters. Finally, it would be interesting, but probably difficult, to design algorithms that find semi-nice tree-decompositions whose Join partitions D, E, F are optimized to give the best worst-case runtime for a particular dynamic programming algorithm. Note that a non-optimal tree-decomposition could in fact be better than an optimal one. A focus on Join nodes with small Expensive sets should be the primary issue.

Acknowledgements. We would like to thank Jochen Alber and Rolf Niedermeier for suggesting the comparison of dynamic programming on tree-decompositions and branch-decompositions [6].

References

1. J. Alber, H. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier, *Fixed parameter algorithms for Dominating Set and related problems on planar graphs*, Algorithmica, 33 (2002), pp. 461–493.
2. J. Alber and R. Niedermeier, *Improved tree decomposition based algorithms for domination-like problems*, in LATIN'02: Theoretical informatics (Cancun), vol. 2286 of Lecture Notes in Comput. Sci., Berlin, 2002, Springer, pp. 613–627.
3. H. Bodlaender, *Treewidth: Algorithmic techniques and results.*, in MFCS'97: Mathematical Foundations of Computer Science 1997, 22nd International Symposium (MFCS), vol. 1295 of Lecture Notes in Computer Science, Springer, 1997, pp. 19–36.
4. W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.
5. E. D. Demaine, F. V. Fomin, M. T. Hajiaghayi, and D. M. Thilikos, *Fixed-parameter algorithms for the (k, r)-center in planar graphs and map graphs*, in ICALP'03: Automata, languages and programming, vol. 2719 of Lecture Notes in Comput. Sci., Berlin, 2003, Springer, pp. 829–844.
6. F. Dorn, *Special branch decomposition: A natural link between tree decompositions and branch decompositions*, Master Thesis, Universität Tübingen, (2004).
7. F. V. Fomin and D. M. Thilikos, *Dominating sets in planar graphs: branch-width and exponential speed-up*, in SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), New York, 2003, ACM, pp. 168–177.
8. T. Kloks, *Treewidth*, vol. 842 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994. Computations and approximations.
9. J. Kratochvil, *Perfect codes in general graphs*, (monograph) Academia Praha, Praha, 1991.
10. N. Robertson and P. Seymour, *Graph minors X. Obstructions to tree-decomposition.*, Journal of Combinatorial Theory Series B, 52 (1991), pp. 153–190.
11. J. A. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Discrete Math, 10 (1997), pp. 529–550.

6 Appendix

Proof of Lemma 1:

Proof: For the timing, the number of necessary triples of indices (s, s_b, s_c) is the product of the number of necessary vertex state triples $(s(v), s_b(v), s_c(v))$ for each vertex in D, E and F, in total $3^{D+F}4^E$. The first step in the computation of $Table_X$ therefore takes time $O(3^{D+F}4^E)$. The remaining steps compute the $New(S)$ sets and update $Table_X$ within the same time bound (to account for the case $|F \cup E| = O(1)$, for each of the $2^F 3^{D+E}$ indices s we compute in step 3 the index s' using $O(1)$ operations on words of length at most 3^X.)

For correctness, assume $Table_B$ and $Table_C$ are correct and consider an index s of $Table_X$. Let the set of new edges be $New = \{uv \in E(G) : \{u, v\} \subseteq D \wedge \{u, v\} \cap B \neq \emptyset \wedge \{u, v\} \cap C \neq \emptyset\}$ and let S be the smallest vertex subset that is legal for the index s in the graph $G_X \setminus New$, i.e. not accounting for the new edges. Let $S_b = S \cap G_B$ and $S_c = S \cap G_C$. Since $G_B \cap G_C = B \cap C$ we have $S_b \cap S_c = S \cap B \cap C$. Note that the subsets S_b and S_c naturally designate indices s_b and s_c in $Table_B$ and $Table_C$ where all vertices in S_b or S_c have state Dom, while the remaining have state Free in case they have no neighbors in S_b or S_c and have state NbrDom otherwise. The triples $s(v), s_b(v), s_c(v)$ thus constructed from S, S_b, S_c are captured by the definition of necessary vertex state triples, except for the possible triple $(s(x) = NbrD, s_b(x) = NbrD, s_c(x) = NbrD)$. We define index s'_b of $Table_B$ by $s'_b(x) = Free$ for vertices x in such a triple just defined and $s'_b(v) = s_b(v)$ for any other vertex. Note that $(s(x) = NbrD, s'_b(x) = Free, s_c(x) = NbrD)$ is a necessary triple. We then have that S_b and S_c are the smallest vertex subsets of G_B and G_C that are legal in $G_X \setminus New$ for the resulting indices s'_b and s_c, by the assumption that S was smallest for index s, and by the monotonicity property $Table_B(s'_b) \leq Table_B(s_b)$. Since s, s'_b, s_c is a necessary triple, the first step of the algorithm will set $Table_X(s)$ to the value $Table_B(s'_b) + Table_C(s_c) - |B \cap C \cap \{v : s(v) = Dom\}| = |S_b| + |S_c| - |S_b \cap S_c| = |S|$.

The second and third steps will account for the edges in New. The only indices s for which the set of legal subsets are not necessarily the same in $G_X \setminus New$ and G_X are those where there exists a new edge ux with $s(u) = Dom$ and $s(x) = NbrD$. For such an index it is possible that a set S is legal in G_X but that the only neighbor x has in S is the new neighbor u so that S is not legal in $G_X \setminus New$. However, S is legal in $G_X \setminus New$ for the index s' where $s'(x) = Free$ for all $x \in D$ with $s(x) = NbrDom$ having a new neighbor u with $s(u) = Dom$, and $s'(v) = s(v)$ otherwise. In case S was the smallest legal subset for s in G_X, the third step of the computation of $Table_X$ would therefore update $Table_X(s) = Table_X(s') = |S|$. $\square$

Sketch of (k, r)-center algorithm:

We can design an algorithm solving the (k, r)-center problem on a semi-nice tree-decomposition by using as basis the algorithm in [5] on branch-decompositions. In the following we mainly describe the algorithm already given by [5], with the only addition being the handling of new edges in steps 2 and 3 of the Join update below. The problem asks whether an input graph G has $\leq k$ vertices, called *centers*, such that every vertex of G is within distance $\leq r$ from some center. We first transform a given branch-decomposition into a semi-nice tree-decomposition as described in Lemma 2. We can design a dynamic programming algorithm on semi-nice tree-decompositions that needs $2r + 1$ states with one state defining centers, r states defining a vertex "having a center at distance i" for each $1 \leq i \leq r$, and r temporary states defining a vertex that "must get a center at distance i" for each $1 \leq i \leq r$ (this center will be reachable by a path through a not-yet considered neighbor.) These latter two types of states obey a monotonicity property similar to the NbrD and Free states. The resulting algorithm will have $O((2r + 1)^{D+F}(3r + 1)^E)$ time for updating Join nodes and $O(X(r+1)^F(2r+1)^{X\setminus F})$ for Introduce nodes X and $O((r+1)^{C\setminus X}(2r+1)^X)$ for Forget node X and child C. On a Join node X with partition D, E, F consider the node t of the branch-decomposition guaranteed by Lemma 2 that has $E \subseteq mid(e) \cap mid(f) \cap mid(g)$ and $F \subseteq mid(f) \cap mid(g) \setminus mid(e)$ and $D \subseteq mid(e) \setminus mid(f) \cap mid(g)$. The first step of the Join update on node X with partition D, E, F can be derived from the update described in [5] for sets $X_3 = mid(e) \cap mid(f) \cap mid(g)$, $X_1 = mid(e) \cap mid(f) \setminus mid(g)$, $X_2 = mid(e) \cap mid(g) \setminus mid(f)$ and $X_4 = mid(f) \cap mid(g) \setminus mid(e)$. We then have to handle the new edges among vertices in D, but we can do that without increase of the runtime. After step 1 we have correct table entries for the graph $G_X \setminus New$, as in the proof of Lemma 1. To account for new edges we compute in step 2 $New(R)$ for each $R \subseteq D$ as in the Min Dom set algorithm, and then in step 3 we update in a loop from $i = 1..r$ each index s by $Table_X(s) = Table_X(s')$ where s' is defined by $s'(u) =$"must get a center at distance i" for any $u \in New(R)$ with $s(u) =$"having a center at distance i" and R being the vertices in D whose state in s is either "must get a center at distance $i-1$" or "having a center at distance $i-1$". Together with a straight-forward extension of the update process on Introduce nodes and Forget nodes we obtain an algorithm on semi-nice tree-decompositions matching the runtime $O(2r+1)^{1.5\,bw}$ of [5] when given a branch-decomposition of width bw.

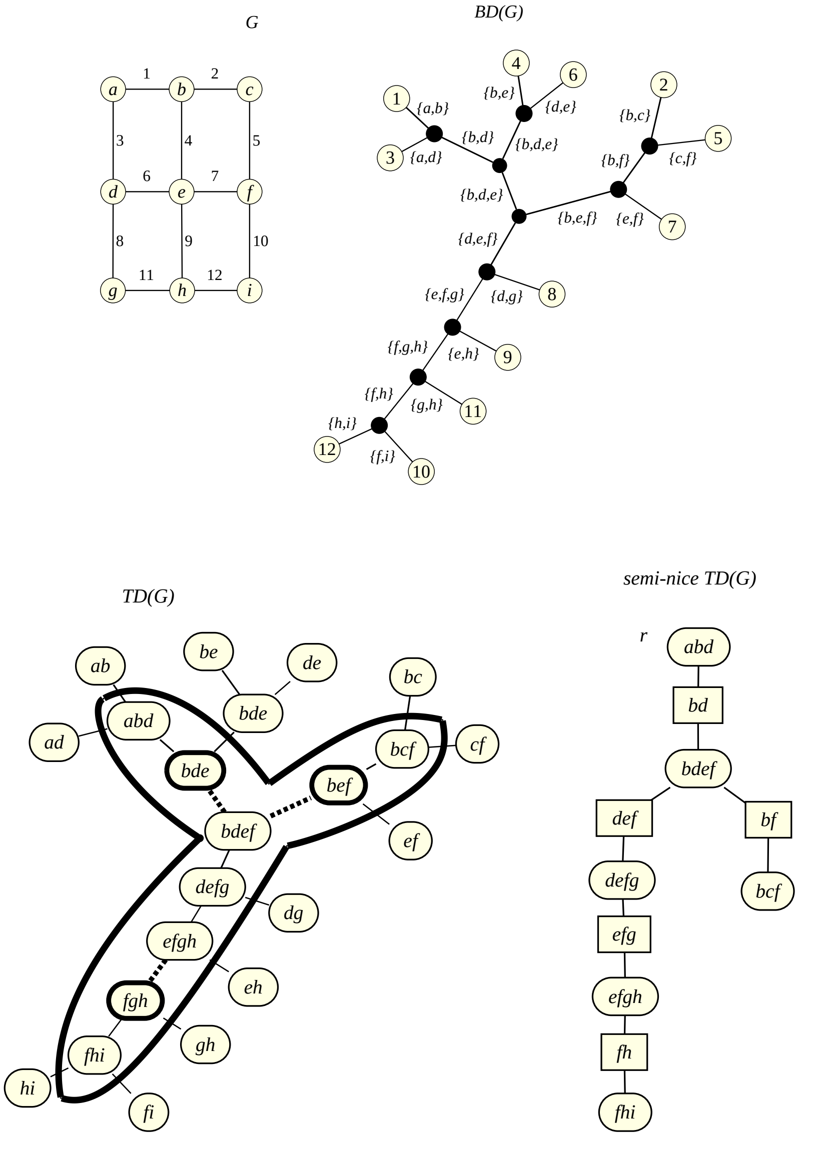

Figure 1. On the upper left a 3×3 grid graph G. On the upper right an optimal branch-decomposition with leaves labeled by edges of G as given by μ and the sets $mid(e)$. On the lower left a tree-decomposition formed in the first step of the algorithm of section 4 with leaf-bags given by μ^{-1} and inner bags given by the union of adjacent $mid(e)$. All nodes outside the bold line are then removed. The edges drawn in a dashed line are contracted and the emphasized bags absorbed by their neighbors. On the lower right the resulting semi-nice tree-decomposition with new nodes emphasized rectangularly and arranged below arbitrary root node r.