Two birds with one stone: the best of branchwidth and treewidth with one algorithm

Frederic Dorn and Jan Arne Telle

Department of Informatics, University of Bergen,
Bergen, Norway

Abstract. In this paper we introduce semi-nice tree-decompositions and show that they combine the best of both branchwidth and treewidth. We first give simple algorithms to transform a given tree-decomposition or branch-decomposition into a semi-nice tree-decomposition. We then give two templates for dynamic programming along a semi-nice tree-decomposition, one for optimization problems over vertex subsets and another for optimization problems over edge subsets. We show that the resulting runtime will match or beat the runtimes achieved by doing dynamic programming directly on either a branch- or tree-decomposition. For example, given a graph G on n vertices with path-, tree- and branch-decompositions of width pw, tw and bw respectively, the Minimum Dominating Set problem on G is solved in time $O(n2^{\min\{1.58\,pw, 2\,tw, 2.38\,bw\}})$ by a single dynamic programming algorithm along a semi-nice tree-decomposition.

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 $bw(G) \leq tw(G) + 1 \leq \frac{3}{2}\, bw(G)$ [16], 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. [5, 15] for overviews. Of the various algorithmic templates suggested for this over the years the nice tree-decompositions [14] 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. Tree-decompositions are in fact moving into the computer science curriculum, *e.g.* twenty pages of a new textbook on Algorithm Design [13] is devoted to this topic.

Recently there have been several papers [10, 7, 6, 12, 11, 8] showing that for graphs of bounded genus the base of the exponent in the running time of these FPT algorithms could be improved by instead doing the dynamic programming along a branch-decomposition of optimal branchwidth. Dynamic programming along either a branch- or tree-decomposition of a graph both share the

property of traversing a tree bottom-up and combining solutions to problems on certain subgraphs that overlap in a bounded-size separator of the original graph. But there are also important differences, *e.g.* the subgraphs mentioned above are for tree-decompositions usually induced by subsets of vertices and for branch-decompositions by non-overlapping sets of edges. Various optimization tricks have been presented to speed up the algorithms, some of these come from the field of tree-decompositions [3, 2] and others from the field of branch-decompositions [10, 7]. As already mentioned it seems that for planar graphs the branchwidth parameter is the better choice, at least for worst-case runtime. There are other graph classes where treewidth is better. In most situations the input graphs contain some graphs where branchwidth is better and others where treewidth is better. If we already have implementations of heuristic algorithms for both branchwidth and treewidth, then the better choice for the dynamic programming stage will rely on the output of these heuristics for a given graph. Both from a theoretical and also applied viewpoint it therefore seems necessary, for each optimization problem, to design and possibly implement two separate dynamic programming algorithms, one for tree-decompositions and another for branch-decompositions. In this paper we show that a single dynamic programming algorithm will suffice to get the best of both treewidth and branchwidth.

For this purpose we introduce semi-nice tree-decompositions that maintain much of the simplicity of the nice tree-decompositions. However, the vertices of a Join are partitioned into 3 sets D,E,F and the binary Join operation treat vertices in each set differently in order to improve runtime. Symmetric Difference vertices D are those that appear in only one of the children, Forget vertices F are those for which all their neighbors have already been considered, and Expensive vertices E are the rest (the formal definitions follow later.) We first show how to transform a given branch-decomposition or tree-decomposition into a semi-nice tree-decomposition. We then give two templates for dynamic programming on semi-nice tree-decompositions, one for vertex subset problems and the other for edge subset problems.

For vertex subset problems we improve the runtime for the Join update operation during dynamic programming. Along the way we also simplify the proof of monotonicity of table entries for domination-type problems of [2] 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 in [3], of lowering to $O(n\lambda^k)$ the runtime of dynamic programming on treewidth k graphs for solving a problem having λ vertex-states. For edge subset problems the two subgraphs for whom solutions are combined in the Join operation are defined to not overlap at all in edges. Edges on vertices common to the two subgraphs are instead introduced in a later Forget operation. In their paper [6] on heuristics for TSP (travelling salesman problem) Cook and Seymour state that when carrying out dynamic programming to solve optimization problems that deal with edge sets branchwidth is a more natural framework than treewidth. We claim that our template shares this property of being a natural framework for edge set problems.

We employ this approach to various problems, such as dominating set problems, some of which had previously been solved for tree-decompositions in [17, 3] and for branch-decompositions in [10], to TSP solved for branch-decompositions in [6] and tree-decompositions in [4], and in the long version to this paper [9] to (k, r)-center solved for branch-decompositions in [7]. In each case we match or improve the running time of the algorithms given in those papers. We do this by combining and extending the various optimization tricks for branchwidth and treewidth used in those papers into our dynamic programming algorithm on semi-nice tree-decompositions. Table 1 gives the resulting worst-case runtime on various domination-type problems that are NP-hard for general graphs. For treewidth the previous best results [3] arise from treating all vertices in the Join as Expensive vertices, thus $tw = E$ in column Join of Table 1 instead of $tw = D+E+F$ as we have. For branchwidth the entry for Minimum Dominating set in the first row of Table 1 matches the previous best [10], while the results for each of the other problems are new. We emphasize that for any problem this is the first time that a single dynamic programming algorithm achieves the best of both treewidth and branchwidth.

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, allthough for each problem there is a cutoff at which pathwidth would have been best.

	States	Join	Total time	tw faster
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	$O(3^{D+F}4^E)$	$O(n2^{\min\{2tw,2.38bw\}})$	$tw \leq 1.19bw$
∃/Min/Max 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	$O(4^{D+F}6^E)$	$O(n2^{\min\{2.58tw,3bw\}})$	$tw \leq 1.16bw$
∃/Min/Max Perf Total Dom	4	$O(4^D5^F6^E)$	$O(n2^{\min\{2.58tw,3.16bw\}})$	$tw \leq 1.22bw$

2 Semi-nice tree-decompositions

We use standard graph notation and terminology, e.g. for a subset $S \subseteq V(G)$ of the vertices of a graph G we let $N(S) = \{v \notin S : \exists u \in S \wedge uv \in E(G)\}$ be 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.

We introduce semi-nice tree-decompositions and two lemmas on transforming a given branch- or tree-decomposition into a semi-nice tree-decomposition. 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 [14] 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** $F = (B \cap C) \setminus A$

D, E, F is a partition of X by definition. Note that if the parent A of $X = B \cup C$ is an Introduce or Join node then $B \cup C \subset A$ and we get $F = \emptyset$. See Figure 1. The *Forgettable* vertices are useful for any node whose parent is a Forget node, and their definition for an Introduce or leaf 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.

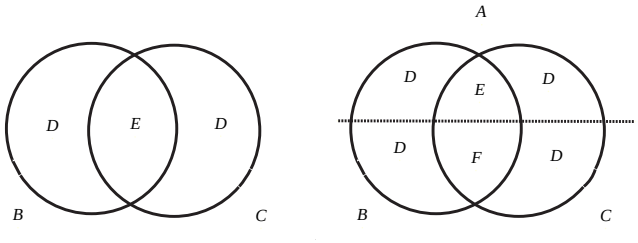

Fig. 1. Two Venn diagrams illustrating the children B,C of a Join node $X = B \cup C$ and its partition D, E, F. On the right the parent A is a Forget node represented by the part of $B \cup C$ above the dashed line. On the left the parent A is not a Forget node and we then have $B \cup C \subset A$ and $F = \emptyset$. In both cases what we call the New edges go between $B \setminus C$ and $C \setminus B$.

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

Note that for a Join node with Forget parent A and children B, C of a sparse semi-nice tree-decomposition any vertex in $B \setminus A \cup C$ has a neighbor in $C \setminus A \cup B$ and vice-versa.

Lemma 1. *Given a tree-decomposition $(T, \mathcal{X})$ of width k of a graph G with n vertices, we can make it into a sparse semi-nice tree-decomposition $(T', \mathcal{X}')$ of width k in time $O(k^2 n)$ while keeping the E-sets in the partition of each Join node as small as the given tree-decomposition allows.*

For proofs see [9]. See Figure 2 for an illustration of the transformation from a given branch-decomposition to a semi-nice tree-decomposition described in the next lemma.

Lemma 2. *Given a branch-decomposition (T, μ) of a graph G with n vertices and m edges we can compute a sparse semi-nice tree-decomposition $(T', \mathcal{X})$ with $O(n)$ nodes in time $O(m)$ 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 $D \subseteq mid(e) \setminus mid(f) \cap mid(g)$.*

3 Dynamic programming for vertex subset problems

In this section we give 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 related to vertex subsets on G. The runtime will in this section be given simply as a function of the D, E, F partition of the Join bags, and $X \setminus F, F$ partition of the other bags. In the final section we will then express the runtimes by pathwidth, branchwidth or treewidth of the graph. We

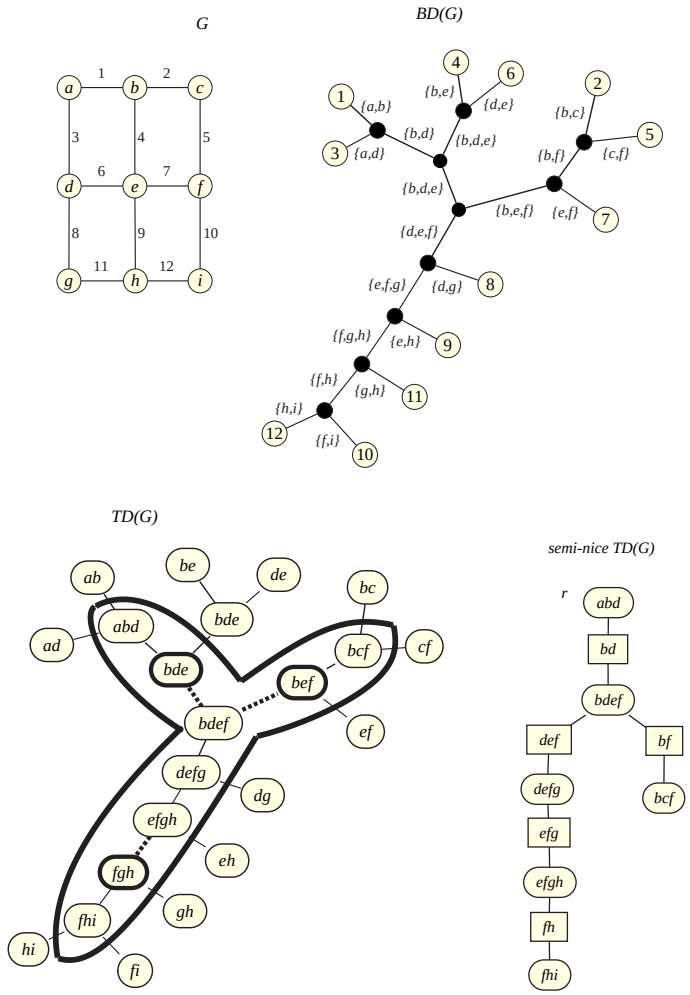

Fig. 2. *Algorithm:* 1) Transform branch-decomposition into a tree-decomposition on the same tree, 2) Transform tree-decomposition into a small tree-decomposition having $O(n)$ nodes, 3) Transform tree-decomposition into a sparse semi-nice tree-decomposition. We illustrate the algorithm with above figure. 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 $mid(e)$. Step 1) is well-known (see e.g. [10] for a correctness proof): On the lower left a tree-decomposition formed with leaf-bags given by μ^{-1} and inner bags given by the union of adjacent $mid(e)$. In step 2) all nodes outside the bold line are then removed. The edges drawn in a dashed line are contracted. For step 3) we apply Lemma 1. On the lower right the resulting semi-nice tree-decomposition with new nodes emphasized rectangularly and arranged below arbitrary root node r.

introduce the template by giving a detailed study of the algorithm for Minimum Dominating sets, and then consider generalizations to various other vertex subset problems like Perfect Code, 2-Packings. We study these variants and the (k, r)-center problem in the long version of this paper where we also give all ommitted proofs [9].

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 $s : X \to \{Dom, NbrD, Free\}$ 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 we have $Table_X(s) = \infty$ 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 $s : X \to \{Dom, NbrD, Free\}$. 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 $O(X 3^{X \setminus F} 2^F)$.

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 three 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. $New = \{uv \in E(G) : u \in B \setminus C \wedge v \in C \setminus B\}$
3. $\forall R \subseteq D : New(R) = \{u \in D \setminus R : \exists v \in R \wedge uv \in New\}$
4. $\forall s : Table_X(s) = Table_X(s')$ 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 [10], 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 3. *The Join update just described for a node X with partition D, E, F is correct and takes time $O(3^{D+F}4^E)$.*

For a proof see [9]. 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 in time $O(n(\max\{4^E 3^{D+F}\} + \max\{X3^{X\setminus F}2^F\}))$ the Min Dominating Set Problem on G 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.*

For problems over vertex subsets having other domination-type constraints we get slightly different runtimes. 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 $|N(v) \cap S| \in \rho$ [17]. Some well-studied and natural types of (σ, ρ)-sets are when σ is either all natural numbers $\mathbb{N}$, all positive numbers $\mathbb{N}^+$, or $\{0\}$, and with ρ being either all positive numbers, or $\{1\}$. The six resulting constraints are Dominating set ($\sigma = \mathbb{N}$, $\rho = \mathbb{N}^+$); Perfect Dominating Set ($\sigma = \mathbb{N}$,$\rho = \{1\}$); Independent Dominating set ($\sigma = \{0\}$,$\rho = \mathbb{N}^+$); Perfect Code ($\sigma = \{0\}$,$\rho = \{1\}$); Total Dominating set ($\sigma = \mathbb{N}^+$, $\rho = \mathbb{N}^+$); Total Perfect Dominating set ($\sigma = \mathbb{N}^+$,$\rho = \{1\}$). 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 thesis [1] considers these six constraints, and give dynamic programming algorithms on nice tree-decompositions that take into account monotonicity properties to arrive at fast runtimes. See column Join in Table 1 for an overview of our results and [9] for exact calculations. The previous best results for these problems [1] 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. These algorithms can of course be extended also to more 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 $V(G) \setminus S$ having at most q neighbors in 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.

4 Dynamic Programming for edge subset problems

Problems like Hamiltonian cycle and Travelling Salesman ask for a subset of edges of the input graph with a given property. An index of the table storing solutions to subproblems will likewise represent a class of edge subsets of the subgraph considered so far. Consider a Join node X with children B, C, and assume that B and C store solutions for the subgraphs G'_B and G'_C. For these edge subset problems the Join operation at X is simplified if we can assume that the two subgraphs G'_B and G'_C do not overlap in edges. To accomplish this we define the subgraph G'_X for edge subset problems to be the graph we get from taking the subgraph G_X as used for vertex subset problems and removing all edges having both endpoints in the set X.

Definition 2. *For edge subset problems the subgraph G'_X of G for which solutions are stored in a table at node X of the tree T is the graph on vertex set $V(G'_X) = \{v \in X' : X' = X$ or X' a descendant of X in $T\}$ and edge set $E(G'_X) = \{uv \in E(G) : \{u, v\} \subseteq V(G'_X)$ and at most one of u and v in $X\}$.*

The implication is that the Join update is simplified, since there is no overlap of edges in the two subgraphs. The Introduce operation becomes trivial, simply adding isolated vertices to the existing subgraph. Likewise, the Initialize-Table operation is trivial since it considers a subgraph without edges. On the other hand the Forget operation becomes more complicated. Let X be a Forget node with child B, thus with $B \setminus X$ the forgotten vertices. Note that an edge between a forgotten vertex $u \in B \setminus X$ and a vertex $v \in X$ has not been considered so far in the algorithm, since it does not belong to G'_B. However, such an edge does belong to G'_X and it will in fact be considered for the first time during the Forget operation at X. This consideration of new adjacencies performed by the Forget operation for edge problems is almost identical to what is performed by the Introduce operation for vertex problems. The Root-Optimization step at root node X becomes trivial since we simply ensure that $|X| = 1$, by a preceding Forget operation.

A comparison with the template given for vertex problems and the one just described shows that for edge problems the Forget-operation is more complicated but the other operations are less complicated. However, note that the gain we get in the runtime of the Join operation for vertex subset problems from the Forgettable vertices F is no longer easily achieved under the edge subset template, since the vertices in F have not had all their adjacencies considered at the time of the Join.

Cook and Seymour [6] give a heuristic algorithm for the Traveling Salesman Problem (TSP). Their paper contains a subroutine which for a subgraph of the input graph solves the TSP problem exactly by dynamic programming along a branch-decomposition. Their paper is not focused on runtime but we can estimate the running time of their dynamic programming algorithm for exact solution of TSP on a heuristically generated branch-decomposition of width k as $O(c^{1.5\,k\,log k}m)$ for some constant c. Their update operation on middle sets of

the branch-decomposition is directly transferred as the update we need for our Join operation, as the subgraphs we are considering do not overlap in edges. When forgetting vertex v we have to consider all neighbors of v in X since these edges have not been accounted for earlier. In the Forget-operation we do this independently for every index of $Table_X$ and every forgotten vertex. Compared to their algorithm, the runtime for our more complicated Forget-operation gives only an additional polynomial factor in the size of the Forget node X. Without going into details in this extended abstract we claim that a dynamic programming algorithm solving TSP on a semi-nice tree-decomposition can in this way be developed exactly as in the paper [6] and with the same exponential runtime.

5 Runtime by branchwidth, treewidth or pathwidth

In this section we assume that we are given a branch-decomposition of width bw or a tree-decomposition of width tw and first transform these into a semi-nice tree-decomposition by the algorithms of Section 2. We then run any of the algorithms described in Sections 3 or 4 to express the runtime to solve those problems as a function of bw or tw. This runtime will match or improve the best results achieved by dynamic programming directly on the branch- or tree-decompositions. For a proof see the long version [9].

Theorem 2. *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* $O(2^{\min\{1.58\,pw, 2\,tw, 2.38\,bw\}})$ *if given all three. For other domination-type problems we get runtimes as in Table 1.*

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 [10] 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 1 for a summary of the results for each domination-type problem. For the TSP problem we have already argued in Section 4 that our algorithm matches the runtime of the algorithm of [6] that works directly on a branch-decomposition.

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

References

1. J. Alber, *Exact algorithms for np-hard problems on networks: Design, analysis, and implementation*, PhD Thesis, Universität Tübingen, (2002).

2. 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.
3. 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 Computer Science, Berlin, 2002, Springer, pp. 613–627.
4. S. Arnborg and A. Proskurowski, *Linear time algorithms for np-hard problems restricted to partial k-trees*, Discrete Applied Math, 23 (1989), pp. 11–24.
5. 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.
6. W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.
7. 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.
8. F. Dorn, E. Penninkx, H. L. Bodlaender, and F. V. Fomin, *Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions*, in ESA'05: 13th Annual European Symposium on Algorithms, vol. 3669 of Lecture Notes in Comput. Sci., Berlin, 2005, Springer, pp. 95–106.
9. F. Dorn and J. Telle, *Two birds with one stone: the best of branchwidth and treewidth with one algorithm*, long version, (2005). `http://www.ii.uib.no/~frederic/DT.pdf`.
10. 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.
11. ———, *Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speed-up*, in ICALP'04:Automata, Languages and Programming: 31st International Colloquium, vol. 3142 of Lecture Notes in Computer Science, Berlin, 2004, Springer, pp. 581–592.
12. ———, *A simple and fast approach for solving problems on planar graphs.*, in STACS'04: 22nd Ann. Symp. on Theoretical Aspect of Computer Science, vol. 2996 of Lecture Notes in Computer Science, Berlin, 2004, Springer, pp. 56–67.
13. J. Kleinberg and E. Tardos, *Algorithm design*, Addison-Wesley, 2005.
14. T. Kloks, *Treewidth*, vol. 842 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994. Computations and approximations.
15. B. Reed, *Treewidth and tangles, a new measure of connectivity and some applications*, Surveys in Combinatorics, 1997.
16. N. Robertson and P. Seymour, *Graph minors X. Obstructions to tree-decomposition.*, Journal of Combinatorial Theory Series B, 52 (1991), pp. 153–190.
17. J. A. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Discrete Math, 10 (1997), pp. 529–550.