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. Branchwidth and treewidth are connectivity parameters of graphs of high importance in algorithm design. By dynamic programming along the associated branch- or tree-decomposition one can solve most graph optimization problems in time linear in the graph size and exponential in the parameter. If one of these parameters is bounded on a class of graphs, then so is the other, but they can differ by a small constant factor and this difference can be crucial for the resulting runtime. 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, a single dynamic programming algorithm along a semi-nice tree-decomposition will solve the Minimum Dominating Set problem on G in time $O(n2^{\min\{1.58\,pw, 2\,tw, 2.38\,bw\}})$. On the applied side the immediate gain is that for each optimization problem one can achieve the benefits of both treewidth, branchwidth and pathwidth while developing and implementing only one dynamic programming algorithm. On the theoretical side the combination of the best properties of both branchwidth and treewidth in a single decomposition is a step towards a more general framework giving the fastest possible algorithms on tree-like graphs.

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 [13] 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 [12] is devoted to this topic.

Recently there have been several papers [9, 7, 6, 11, 10, 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 [9, 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 [9], to (k,r)-center solved for branch-decompositions in [7], and to TSP solved for branch-decompositions in [6] and tree-decompositions in [4]. 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 [9], 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.

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

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 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.

3 Semi-nice tree-decompositions

In this section we introduce semi-nice tree-decompositions and give simple algorithms 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 [13] 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.

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.*

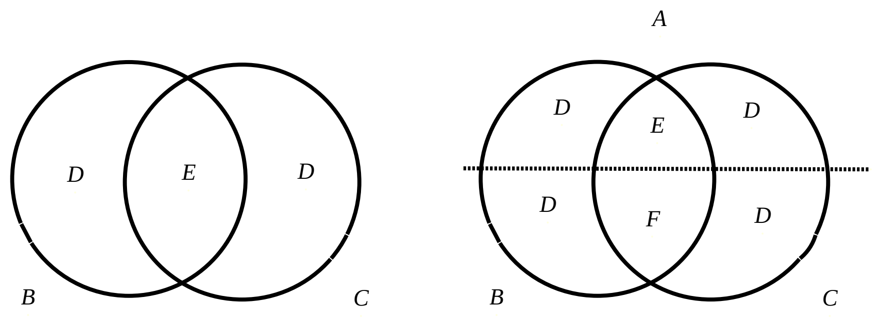

Fig. 1. Two Venn diagrams illustrating the children B,C of a Join node $X = B \cup C$ and its partition D, E, F. On the right the parent A is a Forget node represented by the part of $B \cup C$ above the dashed line. On the left the parent A is not a Forget node and we then have $B \cup C \subset A$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

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 in time $O(k^2n)$ 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.*

Proof. 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. With a Join node X having parent A and $d \geq 2$ children $C_1, C_2, ..., C_d$ turning into a path, every new Join node X_i has an expensive set $E_i \subseteq A \cap C_i \cap (\bigcup_{k=i-1}^{d} C_k)$. This keeps the resulting Expensive sets E as small as possible when we choose the order of $C_1, C_2, ..., C_d$ arbitrarily.

See Figure 2 in the appendix for an illustration of the transformation from a branch-decomposition to a semi-nice tree-decomposition 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. [9] 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 $X \subseteq X_t$.

4 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 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 and (k, r)-center.

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 $O(X 3^{X \setminus F} 2^F)$.

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 [9], 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 $O(3^{D+F}4^E)$.*

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 4 the index s' using $O(1)$ operations on words of length at most 3^X.)

Let the set of new edges be $New = \{uv \in E(G) : u \in B \setminus C \wedge v \in C \setminus B\}$ as in step 2 of the Join. For correctness, assume $Table_B$ and $Table_C$ are correct and consider an index s of $Table_X$. We will first show that in case we had $New = \emptyset$ then step 1 already computes the correct values. Let therefore 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 last three 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 last step of the computation of $Table_X$ would therefore correctly set the value of $Table_X(s)$ to $Table_X(s) = Table_X(s') = |S|$.

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 $O(n(\max\{4^E 3^{D+F}\} + \max\{X 3^{X \setminus F} 2^F\}))$ 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.*

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, all positive numbers, or $\{0\}$, and with ρ being either all positive numbers, or $\{1\}$. The six resulting constraints are 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 even 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 papers [1, 3] considers these six constraints, and give dynamic programming algorithms on nice tree-decompositions that take into account monotonicity properties to arrive at fast runtimes.

For space reasons our algorithms solving these problems by dynamic programming on a semi-nice tree-decomposition have been moved to the appendix. See column Join in Table 1 for an overview of our results. The previous best results for these problems [1, 3] 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 $O(3^D 4^{E+F})$ runtime for the Join operation.

In the paper [7] the (k, r)-center problem is solved by dynamic programming on a branch-decomposition. The problems asks wether an input graph G has at most k vertices, called centers, such that every vertex is within distance at most r from some center. 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 [7] 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 [7]. For space reasons the sketch of the algorithm is put in the appendix.

5 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 $V(G'_X) = \{v \in X' : X' = X \text{ or } X' \text{ a descendant of } X \text{ in } T\}$ and edge set $E(G'_X) = \{uv \in E(G) : \{u,v\} \subseteq V(G'_X) \text{ and at most one of } u \text{ and } v \text{ in } X\}$.*

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.

6 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 3. We then run any of the algorithms described in Sections 4 or 5 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.

Theorem 2. *We 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.*

Proof. We argue in detail only for the Minimum Dominating Set problem, as the other problems in Table 1 are handled by completely analogous arguments. Given a branch-decomposition (T, μ) of width bw we first transform it into a sparse semi-nice tree-decomposition $(T', \mathcal{X})$ by the algorithm of Lemma 2. We then apply 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.

If given a tree-decomposition of width tw we first transform it into a sparse semi-nice tree-decomposition using Lemma 1. The worst-case occurs for a Join node with $tw + 1$ Expensive vertices, and Theorem 1 then gives runtime $O(2^{2\,tw}n)$. Note that when applying the algorithm of Lemma 1 to a path-decomposition of width pw (which of course is also a tree-decomposition of width pw) the resulting sparse semi-nice tree-decomposition will not have any Join nodes. The worst-case runtime then occurs for an Introduce or Root-optimization 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 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 [9] working directly on the branch-decomposition (except that the runtime there is expressed with multiplicative factor m instead of our n but we have $m = O(n\,bw)$.) See Table 1 for a summary of the results for each domination-type 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.

For the (k, r)-center problem we get an algorithm with runtime $O(2r + 1)^{1.5\,bw}$, see the details in the Appendix, matching the time achieved by [7] for an algorithm working directly on the branch-decomposition. For the TSP problem we have already argued in Section 5 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 on tree-decompositions and branch-decompositions.

References

1. J. Alber, *Exact algorithms for np-hard problems on networks: Design, analysis and implementation*, PhD Thesis, Univ Tubingen, (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. A. 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, 2005, p. to appear.
9. 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.
10. ———, *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.
11. ———, *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.
12. J. Kleinberg and E. Tardos, *Algorithm design*, Addison-Wesley, 2005.
13. T. Kloks, *Treewidth*, vol. 842 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994. Computations and approximations.
14. J. Kratochvil, *Perfect codes in general graphs*, (monograph) Academia Praha, Praha, 1991.
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.

7 Appendix

Sketch of proof for vertex subset problems in Table 1

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 [14]. 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 $s : X \rightarrow \{Dom, 0NbrD, 1NbrD\}$ 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 3.

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 $s : X \rightarrow \{DomNbrD, NbrD, DomFree, Free\}$ 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

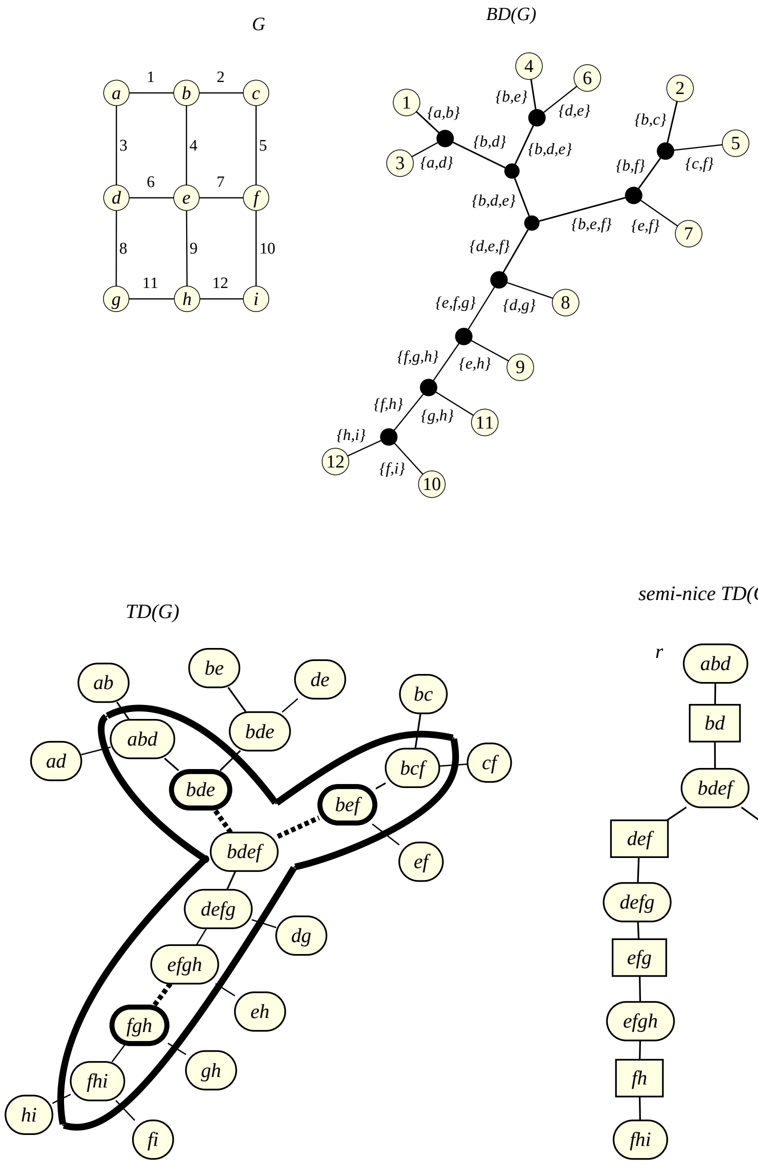

Fig. 2. On the upper left a 3×3 grid graph G. On the upper right an optimal branch-decomposition with leaves labeled by edges of G as given by μ and the sets $mid(e)$. On the lower left a tree-decomposition formed in the first step of the algorithm of section 6 with leaf-bags given by μ^{-1} and inner 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.

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 [3] incorrectly claims runtime $O(5^k)$, but this has been corrected to $O(6^k)$ in [1]. 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 $O(4^D5^F6^E)$.

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 [7] on branch-decompositions. In the following we mainly describe the algorithm already given by [7], with the only addition to that algorithm being the handling of new edges in 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 [7] 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 3. To account for new edges we compute in step 2 and 3 New and $New(R)$ for each $R \subseteq D$ as in the Min Dom set algorithm, and then in step 4 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 [7] when given a branch-decomposition of width bw.