Improved Algorithms and Complexity Results for Power Domination in Graphs[*]

Jiong Guo[†] Rolf Niedermeier[†] Daniel Raible[‡]

July 3, 2007

Abstract

The NP-complete Power Dominating Set problem is an "electric power networks variant" of the classical domination problem in graphs: Given an undirected graph $G = (V, E)$, find a minimum-size set $P \subseteq V$ such that all vertices in V are "observed" by the vertices in P. Herein, a vertex observes itself and all its neighbors, and if an observed vertex has all but one of its neighbors observed, then the remaining neighbor becomes observed as well. We show that Power Dominating Set can be solved by "bounded-treewidth dynamic programs." For treewidth being upper-bounded by a constant, we achieve a linear-time algorithm. In particular, we present a simplified linear-time algorithm for Power Dominating Set in trees. Moreover, we simplify and extend several NP-completeness results, particularly showing that Power Dominating Set remains NP-complete for planar graphs, for circle graphs, and for split graphs. Specifically, our improved reductions imply that Power Dominating Set parameterized by $|P|$ is W[2]-hard and it cannot be better approximated than Dominating Set.

Key Words. Design and analysis of algorithms, computational complexity, parameterized complexity, fixed-parameter algorithms, graph algorithms, graphs of bounded treewidth, (power) domination in graphs.

1 Introduction

Domination is a central theme in graph theory [23, 22, 24]. The basic problem is: given an undirected graph $G = (V, E)$, determine a minimum-size vertex set

[*]A preliminary version of this paper appears in the proceedings of the 15th International Symposium on Fundamentals of Computation Theory (FCT'05), volume 3623 in LNCS, pages 172–184, Springer-Verlag, 2005. Work supported by the Deutsche Forschungsgemeinschaft (DFG), Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4. Significant portions of this work were done while all authors were affiliated with the Universität Tübingen.

[†]Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany. {guo,niedermr}@minet.uni-jena.de

[‡]Abteilung Informatik/Wirtschaftsinformatik, Fachbereich IV, Universität Trier, D-54286 Trier, Germany. raible@uni-trier.de

$D \subseteq V$ such that each vertex v is contained in D or v is a neighbor of at least one vertex in D. The corresponding decision problem DOMINATING SET (DS) is NP-complete and W[2]-complete [16, 30]. Unless NP-hard problems have slightly superpolynomial-time algorithms, DOMINATING SET is not polynomial-time approximable better than $\Theta(\log |V|)$ [17]. Numerous variations of DOMINATING SET exist. For instance, CONNECTED DOMINATING SET—which recently has received particular interest for routing in ad-hoc networks [2]—additionally requires that the dominating set D induces a connected subgraph of G. Opposite to DOMINATING SET, CONNECTED DOMINATING SET carries some form of *non-locality*: the correctness of the connected dominating set cannot be decided by locally checking every direct neighborhood. In this work, we study another "non-local" variant of domination which appears in electric power networks [12, 21]—POWER DOMINATING SET (PDS). This variant is motivated by monitoring an electric power network, where one is asked to place a minimum number of so-called *phase measurement units* (PMU) at some locations in the system to measure the state variables (for example, the voltage magnitude). Since the monitoring PMU devices are expensive, to save costs one naturally comes to the minimization problem modelled by PDS.

Intuitively, in comparison with CONNECTED DOMINATING SET we have a more complex degree of non-locality in PDS. Whereas CONNECTED DOMINATING SET only refers to a property of the solution set, PDS has the non-locality in its domination mechanism: A vertex may dominate vertices at *arbitrary* distance when certain conditions are fulfilled. For DS, we have one "observation rule" concerning vertices in the dominating set: these vertices observe (dominate) themselves and all their neighbors (and nothing else). The goal is to get all vertices observed by a minimum number of observers. By way of contrast, we have an additional observation rule in the case of PDS. This rule says that for an already observed vertex whose all but one neighbors are already observed, the one remaining neighbor becomes observed as well.[1] Note that the second observation rule brings in non-locality. The graph-theoretical and algorithmic study of PDS was initiated by Haynes et al. [21] and has been further pursued for special cases [12, 15, 29]. Haynes et al. showed that PDS is NP-complete for general graphs as well as for bipartite and chordal graphs. Moreover, they presented a linear-time algorithm to solve PDS in trees. We improve on their results in the following ways.

- We present simplified and "stronger" NP-completeness proofs, (reduction from DOMINATING SET instead of reduction from 3-SATISFIABILITY), giving all results of Haynes et al. in a significantly simplified way and additionally implying NP-completeness also for planar graphs, circle graphs, and split graphs. Moreover, our simple reductions preserve parameterized complexity [16, 18, 30] and polynomial-time approximability [7, 35]. So we can conclude that, in case of general graphs, PDS is W[2]-hard and

[1] This is motivated by Kirchhoff's law in electrical network theory. The original definition of Haynes et al. [21] is a bit more complicated and the equivalent definition presented here is due to Kneis et al. [27].

it is only $\Theta(\log |V|)$-approximable unless unlikely collapses in structural complexity theory occur.[2]

- We present a simpler linear-time algorithm for PDS in trees than the one presented by Haynes et al. [21].
- We develop a concrete dynamic programming algorithm for PDS in graphs of bounded treewidth, answering an open question of Haynes et al. [21]. In fact, for treewidth being upper-bounded by a constant this gives a linear-time algorithm for PDS. To this end, a crucial contribution is the introduction of the concept of "valid orientations" of undirected graphs. Independently, fixed-parameter tractability with respect to parameter treewidth was also shown by Kneis et al. [27] using descriptive complexity tools. They express PDS in monadic second-order logic[3], which implies algorithms of highly super-exponential running time. In contrast, we develop a direct algorithm for PDS which can be described and analyzed by standard means without the "unimplemented" formalism of monadic second-order logic that implicitly contains a huge overhead. Moreover, our new concept of valid orientations allows for concrete studies of PDS in further contexts such as directed graphs [1].

Let us return to the issue of non-locality. Demaine and Hajiaghayi [13], answering an open question from Alber et al. [3], showed that Connected Dominating Set can be solved by bounded-treewidth dynamic programs. It was not believed before [3] that such non-local properties as "connectedness" could be captured in this way. In case of PDS, the "even worse" degree of non-locality appears to make things still harder. Nevertheless, a concrete dynamic programming solution can be found as we demonstrate here. Finally, a somewhat different contribution of this work is to initiate the comparison between classical DS and PDS in terms of algorithmic complexity (to some extent already taken up by Aazami and Stilp [1] in terms of polynomial-time approximability). In particular, Table 1 in Section 3 leads to several issues for future research.

2 Preliminaries

We assume basic familiarity with fundamental concepts of algorithmics and graph theory. All graphs $G = (V, E)$ in this work are simple and without self-loops. With $V(G)$ and $E(G)$, we denote the vertex set V and the edge set E of a graph $G = (V, E)$, respectively. For a vertex v in graph G, we denote by $N_G(v)$ the open neighborhood of v in G. By $N_G[v]$, we refer to the closed neighborhood of v. This naturally generalizes to $N_G(U)$ and $N_G[U]$ for U being

[2]Basically the same reduction was independently found by Kneis et al. [27]. Independently, Liao and Lee [29] also have shown the NP-completeness of PDS on split graphs by giving a reduction from 3SAT.

[3]This confirmed a conjecture made by Petr Hliněný in a discussion with Peter Rossmanith and Rolf Niedermeier at the IWPEC 2004 meeting in Bergen, Norway, September 2004.

a set of vertices. A subgraph of G induced by a vertex subset $V' \subseteq V$ is denoted by $G[V']$.

For notions concerning approximation algorithms we refer to [7, 35], for parameterized complexity we refer to [16, 18, 30], and for NP-completeness theory we refer to the classic of Garey and Johnson [19].

To define power domination in graphs, we use two (simplified) *observation rules* due to Kneis et al. [27].

- **Observation Rule 1 (OR1)**: A vertex in the power domination set observes itself and all of its neighbors.
- **Observation Rule 2 (OR2)**: If an observed vertex v of degree $d \geq 2$ is adjacent to $d-1$ observed vertices, then the remaining unobserved neighbor becomes observed as well.

In this way, we arrive at the definition of the central problem of this work:

> Power Dominating Set (PDS)
> **Input**: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.
> **Question**: Is there a set $P \subseteq V$ with $|P| \leq k$ which observes all vertices in V with respect to the two observation rules OR1 and OR2?

Herein, P is called a *power dominating set* of G. The classical Dominating Set (DS) problem can be defined by simply omitting OR2. The following Lemma is due to Haynes et al. [21].

Lemma 1. *[21, Observation 4] If G is a connected graph with at least one vertex of degree three or higher, then there is always a minimum power dominating set which only contains vertices with degree at least three.*

Haynes et al. [21] also showed that, given an arbitrary power dominating set P for a connected graph with at least one vertex of degree three or higher, one can construct in linear time a power dominating set P' with $|P'| \leq |P|$ such that P' only contains vertices with degree at least three.

The following property of a minimum power dominating set will be used by the dynamic programming algorithm in Section 4.2.2.

Lemma 2. *Given a connected graph $G = (V, E)$ with $|V| > 2$, there always exists a minimum power dominating set P of G such that, for each vertex $u \in P$, $|N(u) \setminus N[P \setminus \{u\}]| \geq 2$.*

Proof. Given a vertex set P and a vertex $u \in P$, let $N(u, P) := N(u) \setminus N[P \setminus \{u\}]$ and use $\alpha(P)$ to denote the number of the vertices $u \in P$ with $|N(u, P)| \leq 1$. We prove the lemma by showing that each minimum power dominating set P with $\alpha(P) \geq 1$ can be transformed into a new minimum power dominating set P' with $\alpha(P') = \alpha(P) - 1$.

Let $u \in P$ with $|N(u, P)| \leq 1$. Due to Lemma 1 we can assume that $|N(u)| \geq 3$. First, we show that $u \notin N(P \setminus \{u\})$. Suppose that this is not true.

Then $N[u]$ is observed by $P \setminus \{u\}$: in the case $|N(u, P)| = 0$, this is clear; in the case $|N(u, P)| = 1$, apply once OR2 to u. This implies that $P \setminus \{u\}$ is a power dominating set as well, a contradiction to the minimality of P.

Next, we show that there exists a vertex $v \in N(u)$ with $|N(v) \setminus N[P \setminus \{u\}]| \geq 2$. Suppose that this is not true. From $u \notin N(P \setminus \{u\})$, we conclude that $N(v) \setminus N[P \setminus \{u\}] \subseteq \{u\}$ for all $v \in N(u)$. In both bases, $|N(u, P)| = 0$ and $|N(u, P)| = 1$, we can find a vertex $v \in N(u)$ that is observed by $P \setminus \{u\}$, and the vertices in $N[v]$, with the only exception of u, are observed by $P \setminus \{u\}$ as well. Then, u gets observed by applying OR2 to v. If $|N(u, P)| = 1$, then another application of OR2 to u makes $N[u]$ observed. This means that $P \setminus \{u\}$ is a power dominating set, again a contradiction to the minimality of P.

After all, we can assume that there exists a vertex $v \in N(u)$ with $|N(v) \setminus N[P \setminus \{u\}]| \geq 2$. Then, we can construct a new power dominating set for G by setting $P' := P \cup \{v\} \setminus \{u\}$. Note that v has at least two neighbors outside of $N[P' \setminus \{v\}]$, one of them being u. This completes the proof of the lemma. □

The central concept of this work are tree decompositions of graphs and their use with respect to dynamic programming as, e.g., described in [8, 9, 26, 32].

Definition 1. Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $\langle \{X_i : i \in I\}, T \rangle$, where each X_i is a subset of V, called a *bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:

1. $\bigcup_{i \in I} X_i = V$;
2. for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$;
3. for all $i, j, k \in I$, if j lies on the path between i and k in T, then $X_i \cap X_k \subseteq X_j$.

The *width* of $\langle \{X_i : i \in I\}, T \rangle$ equals $\max\{|X_i| : i \in I\} - 1$. The *treewidth* of G is the minimum k such that G has a tree decomposition of width k.

The third property is called the *consistency property* of tree decompositions. By this definition, a tree is nothing but a graph with treewidth one. To simplify the development of dynamic programs we consider tree decompositions with a particularly simple structure:

Definition 2. A tree decomposition $\langle \{X_i : i \in I\}, T \rangle$ is called *nice* if T is rooted and the following conditions are satisfied:

1. Every node of the tree T has at most 2 children.
2. If a node i has two children j and k, then $X_i = X_j = X_k$ (in this case i is called a JOIN NODE).
3. If a node i has one child j, then one of the following holds:

   (1) $|X_i| = |X_j| + 1$ and $X_j \subset X_i$ (in this case i is called an INTRODUCE NODE),

(2) $|X_i| = |X_j| - 1$ and $X_i \subset X_j$ (in this case i is called a FORGET NODE).

Every tree decomposition can be efficiently transformed into a nice tree decomposition:

Lemma 3. *[26, Lemma 13.1.3] Given a graph $G = (V, E)$ together with an $O(|V|)$-nodes width-k tree decomposition, one can find in $O(|V|)$ time a nice tree decomposition of G that also has width k and $O(|V|)$ nodes.*

In Section 3, we consider the following three graph classes.

Definition 3. A graph G is *chordal* if each cycle in G of length at least four has at least one *chord*. A chord of a cycle is an edge between two vertices of the cycle that is not an edge of the cycle.

A graph G is a *circle* graph if G is the intersection graph of chords in a cycle.

A graph G is a *split* graph if there is a partition of its vertex set into a clique and an independent set.

We refer to Brandstädt et al. [11] for a survey on graph classes.

3 Complexity Results

Haynes et al. [21] showed the NP-completeness of PDS by giving a reduction from 3-SAT. This also gives that PDS remains NP-complete when the input graph is bipartite or chordal. As DS is the same as PDS without applying OR2, a natural question arises: does OR2 make PDS more difficult to solve than DS? Intuitively, one would say yes due to the non-locality of this rule. In this section, we give some first evidence for this intuition, that is, we show, for general graphs, that PDS is at least as hard to solve as DS by reducing DS to PDS. We remark that our reduction from DS to PDS is much simpler than the one from 3-SAT to PDS. Moreover, our reduction implies the NP-completeness of PDS also in case of bipartite, chordal, circle, and planar graphs. Polynomial-time inapproximability and parameterized intractability results for DS transfer to PDS along the same lines.

Theorem 1. POWER DOMINATING SET *is NP-complete in bipartite, chordal, circle, and planar graphs.*

Proof. Since it can be easily decided whether a vertex set P is a power dominating set, PDS is in NP. To show NP-hardness, we give a reduction from DS.

Given a DS-instance with $G = (V, E)$ and the parameter k, we construct a PDS-instance with $G' = (V \cup V_1, E \cup E_1)$ and the same parameter k: attach newly introduced degree-one vertices to all vertices from V. Figure 1 illustrates this transformation.

Let D be a dominating set of G. We show that D is a power dominating set of G'. By definition, all vertices from V are observed by OR1. Applying OR2 to every vertex in V, the vertices in V_1 become observed as well. Thus, D is a power dominating set of G'.

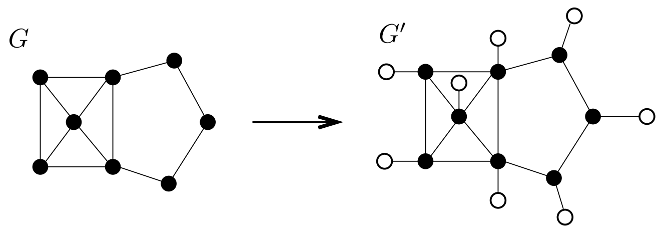

Figure 1: An example of the reduction from DS to PDS in the proof of Theorem 1. The vertices in V_1 are drawn white.

If G' has a power dominating set P, then we can assume due to Lemma 1 that each vertex of P has degree at least three. This implies that $P \cap V_1 = \emptyset$. In order to observe a vertex $v \in V_1$, OR1 or OR2 has to be applied to v's only neighbor $u \in V$. This means that either $u \in P$ or u's neighbors in G cannot be observed by applying OR2 to u. Hence the vertices in $V \setminus P$ must be observed by applying OR1 to one of their neighbors in G. This means that P is a dominating set for G.

It is easy to verify that the reduction preserves the graph properties "bipartite," "chordal," "circle," and "planar." Hence, the NP-completeness of PDS follows from the NP-completeness of DS for bipartite graphs [14], chordal graphs [10], circle graphs [25], and planar graphs [19]. $\square$

The reduction in the proof of Theorem 1 was independently achieved by Kneis et al. [27]. Observe that it is a gap-preserving as well as a parameterized reduction. This implies that all negative results with respect to the approximability and parameterized tractability with respect to the solution size for DS also are valid for PDS. As a consequence, it is hard to polynomial-time approximate PDS better than $\Theta(\log |V|)$ [17] and PDS is W[2]-hard with the size of the power dominating set as parameter [16].

Next, we show that PDS is NP-complete in split graphs.[4] The reduction is from the NP-complete Vertex Cover problem [19]: Given an undirected graph $G = (V, E)$ and an integer $k \geq 0$, is there a set $C \subseteq V$ with $|C| \leq k$ such that each edge has at least one endpoint in C?

Theorem 2. Power Dominating Set *is NP-complete in split graphs.*

Proof. For a Vertex Cover instance with graph $G = (V, E)$, we construct a split graph G' from G as follows. For each edge $e = \{u, v\} \in E$ we add a new vertex w_e and two edges between w_e and u and between w_e and v to G. We denote the set of the vertices w_e by V_E. Moreover, we introduce for each vertex $v \in V$ a new degree-one vertex v' and an edge between v and v'. The set

[4]A reduction from 3SAT to PDS in split graphs has been described by Liao and Lee [29]. Note that the reduction given in the proof of Theorem 2 is a gap-preserving reduction. This implies that PDS is MaxSNP-hard in split graphs which does not follow from the previous results [29].

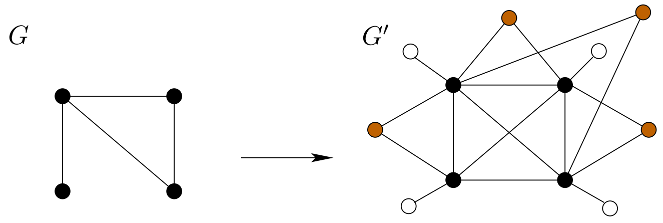

Figure 2: An example of the reduction from Vertex Cover to PDS in the proof of Theorem 2. The vertices in V_E and V_1 are drawn grey and white, respectively.

of these vertices is denoted by V_1. Finally, we complete the graph induced by V into a clique. Note that the subgraph of G' induced by the vertices in $V_E \cup V_1$ is an independent set. Thus, G' is a split graph. See Figure 2 for an example. We claim that G has a vertex cover of size k iff G' has a power dominating set of size k. The proof of the claim uses a similar argument as in the proof of Theorem 1 [31]. □

In summary, Table 1 compares the computational complexity of PDS and DS. By means of Theorems 1 and 2 we now have established NP-completeness results for PDS on all graph classes where NP-completeness is listed for DS by Kratsch [28].[5] Liao and Lee [29] showed that PDS is polynomial-time solvable in interval graphs.

4 Dynamic Program for Graphs of Bounded Treewidth

We have seen in the previous section that PDS is W[2]-hard with respect to the parameter k denoting the size of the power dominating set. This means that, in general, there is basically no hope for an algorithm solving PDS in $f(k) \cdot n^{O(1)}$ time, that is, restricting the combinatorial explosion only to the parameter k (herein, $f(k)$ is a computable function only depending on the parameter k and may be exponentially growing or even worse). By way of contrast, only trivial $n^{O(k)}$ algorithms seem feasible for PDS. Hence, the natural question arises whether we can restrict the combinatorial explosion to other useful parameters. In fact, we show here that we can do so. More specifically, we show that PDS can be solved in $f(k) \cdot n$ time when k denotes the treewidth of the underlying graph. In other words, PDS is fixed-parameter tractable with respect to treewidth. Our main result in the following is to show that, when given a tree decomposition of bounded width for the underlying graph, PDS can be solved in linear time. To this end, we develop a concrete dynamic program using tree decompositions of graphs (Section 4.2). Before that, we start with the simple

[5]Note that the class of comparability graphs is a superclass of the class of bipartite graphs [11]. The NP-completeness of PDS for this graph class directly follows from the NP-completeness of PDS for bipartite graphs.

Table 1: The first column is taken from Kratsch [28]. Partial k-trees are the same as graphs of treewidth k; the linear-time result for partial k-trees with fixed k is shown in Section 4. The result for interval graphs is due to Liao and Lee [29]. Empty entries mean that this has not been studied yet.

Graph classes	Dominating Set	Power Dominating Set
bipartite	NP-complete	NP-complete
chordal	NP-complete	NP-complete
circle	NP-complete	NP-complete
comparability	NP-complete	NP-complete
planar	NP-complete	NP-complete
split	NP-complete	NP-complete
AT-free	poly. time	
cocomparability	poly. time	
distance hereditary	poly. time	
dually chordal	linear time	
interval	poly. time	poly. time
k-polygon ($k \geq 3$)	poly. time	
partial k-tree ($k \geq 1$)	linear time	linear time
permutation	linear time	
strongly chordal	poly. time	

special case of trees (Section 4.1). These results improve previous results from the literature [21, 27].

4.1 Trees

Haynes et al. [21] gave a linear-time algorithm for PDS in trees. As a warm-up for our algorithm working on graphs of bounded treewidth, we start with a much simpler linear-time algorithm for these "treewidth-one graphs." Without loss of generality, we assume that the input tree T is rooted at a degree-one vertex r and the *depth* of a vertex v is defined as the length of the path between v and r. The algorithm follows a bottom-up strategy, see Figure 3.

Theorem 3. *Algorithm PDS-Trees from Figure 3 solves* Power Dominating Set *in trees in linear time.*

Proof. First, we prove that the output P of Algorithm PDS-Trees is a power dominating set. The proof is based on an induction on the depth of the vertices u in T, denoted by depth(u). Note that the proof works top-down whereas the algorithm works bottom-up. For depth(u) $= 0$, it is clear that u which is r is observed due to the "if"-condition on line 9 of Algorithm PDS-Trees. Suppose that all vertices u with depth(u) $< k$ with $k > 0$ are observed. Consider a vertex u with depth(u) $= k$ which is a child of the vertex v. If $v \in P$, then u is observed; otherwise, due to the induction hypothesis, v is observed.

```
PDS-Trees
Input:    Rooted tree $T$ with $r$ denoting the root
Output:   Minimum power dominating set $P$
1     Sort the inner nodes of $T$ in a list $L$ according to a post-order traversal of $T$;
2     while $L \neq \{r\}$ do
3         $v \leftarrow$ the first node in $L$; $L \leftarrow L \setminus \{v\}$;
4         if $v$ has at least two unobserved children then
5             $P \leftarrow P \cup \{v\}$;
6             Exhaustively apply the two observation rules to $T$;
7         end if
8     end while
9     if $r$ is unobserved then
10        $P \leftarrow P \cup \{r\}$;
11    end if
12    return $P$
```

Figure 3: Algorithm to compute a minimum power dominating set of a tree.

Moreover, if $\text{depth}(v) \geq 1$, then v's parent is observed as well. Because of the "if"-condition on line 4 of Algorithm PDS-Trees, v is not in P only if v has at most one unobserved child during the "while"-loop (line 2) of Algorithm PDS-Trees processing v. If vertex u is this only unobserved child, then it gets observed by applying OR2 to v, because v itself and all its neighbors (including v's parent and its children) with the only exception of u are observed by the vertices in P. In summary, we conclude that all vertices in T are observed by P.

Next, we prove the optimality of P by showing a more general statement:

Claim. Given a rooted tree $T = (V, E)$ with root r, Algorithm PDS-Trees outputs a power dominating set P such that $|P \cap V_u| \leq |Q \cap V_u|$ for any optimal solution Q of PDS for T and any tree vertex u, where V_u denotes the vertex set of the subtree of T rooted at u.

Proof of the Claim. We use $T_u = (V_u, E_u)$ to denote the subtree of T rooted at vertex u. Let l denote the depth of T, that is, $l := \max_{v \in V}\{\text{depth}(v)\}$. For each vertex $u \in V$, we define $p_u := |P \cap V_u|$ and $q_u := |Q \cap V_u|$. We use an induction on the depth of tree vertices to prove the claim, starting with the maximum depth l and proceeding to 0.

Since vertices u with $\text{depth}(u) = l$ are leaves and Algorithm PDS-Trees adds no leaf to P, then we have $p_u = 0$ and thus $p_u \leq q_u$ for all u with $\text{depth}(u) = l$.

Suppose that $p_u \leq q_u$ holds for all u with $\text{depth}(u) > k$ with $k < l$. Consider a vertex u with $\text{depth}(u) = k$. Let C_u denote the set of u's children. Then, for all $v \in C_u$, by the induction hypothesis, we have $p_v \leq q_v$. In order to show $p_u \leq q_u$, we only have to consider the case that

$$(a)\, u \in P, \quad (b)\, u \notin Q, \quad \text{and } (c) \sum_{v \in C_u} p_v = \sum_{v \in C_u} q_v. \tag{1}$$

In all other cases, $p_u \leq q_u$ always holds. In the following, we will show that this case does not apply. We assume that $u \neq r$. The argument works also for $u = r$.

From (c) and, for all $v \in C_u$, $p_v \leq q_v$ (induction hypothesis), we know that $p_v = q_v$ for all $v \in C_u$. Moreover, (a) is true only if u has two unobserved children v_1 and v_2 during the "while"-loop of Algorithm PDS-Trees processing u (the "if"-condition on line 4). In other words, this means that vertices v_1 and v_2 are not observed by the vertices in $(P \cap V_u) \setminus \{u\}$. In the following, we show that u has to be included in Q in order for Q to be a valid power dominating set. To this end, we need the following:

$$\forall\, x \in (Q \cap V_{v_i}),\ 1 \leq i \leq 2: \quad \exists x' \in W_{(v_i,x)} : x' \in P \tag{2}$$

where $W_{(v_i,x)}$ denotes the path between v_i and x (including v_i and x).

Without loss of generality we consider only $i = 1$. Assume that predicate (2) is not true for a vertex $x \in Q \cap V_{v_1}$. Since $x \notin P$, we can infer that $p_x < q_x$. Since no vertex from $W_{(v_1,x)}$ is in P and, by induction hypothesis, $p_y \leq q_y$ for all $y \in V_{v_1}$, it follows that $p_{x'} < q_{x'}$ for all vertices $x' \in W_{(v_1,x)}$, in particular, $p_{v_1} < q_{v_1}$. This contradicts the fact that $p_{v_i} = q_{v_i}$ for all of u's children v_i. Thus, we have shown that predicate (2) is true.

By predicate (2), for each vertex $x \in Q \cap V_{v_i}$ ($i \in \{1, 2\}$), there exists a vertex $x' \in P$ on the path $W_{(v_i,x)}$. Thus, if vertex v_i is observed by a vertex $x \in Q \cap V_{v_i}$, then there is a vertex $x' \in P \cap W_{(v_i,x)}$ that observes v_i. However, (a) in (1) is true only if v_1 and v_2 are unobserved by the vertices in $P \cap V_{v_1}$ and $P \cap V_{v_2}$. Altogether, this implies that v_1 and v_2 cannot be observed by the vertices in $Q \cap V_{v_1}$ and $Q \cap V_{v_2}$. Since Q is a solution for PDS in T, we have to take u into Q; otherwise, u has two unobserved neighbors v_1 and v_2. OR2 can never be applied to u whatever vertices from $V \setminus V_u$ are in Q. We can conclude that the case that $u \in P$, $u \notin Q$, and $\sum_{v \in C_u} p_v = \sum_{v \in C_u} q_v$ does not apply. This completes the proof of the claim.

Concerning the running time, we explain how to implement the exhaustive application of OR1 and OR2 (line 6 in Figure 3). Observe that, if OR2 is applicable to a vertex u during the bottom-up process, then all vertices in T_u can be observed at the current stage of the bottom-up process. In particular, after adding vertex u to P, all vertices in T_u can be observed. Thus, exhaustive application of OR1 and OR2 to the vertices of T_u can be implemented as pruning T_u from T which can be done in constant time. Next, consider the vertices in $V \setminus V_u$. After adding u to P, the only possible application of OR1 is that u observes u's parent v. Moreover, we check the applicability of OR2 to the vertices x lying on the path from u to r, in the order of their appearance, the first v, and the last r. As long as OR2 is applicable to a vertex x on this path, we delete x from the list L that is defined in line 1 of Algorithm PDS-Trees, and prune T_x from T.

The linear running time of the algorithm is then easy to see: The vertices to which OR2 is applied are removed from the list L immediately after the application of OR2 and are never processed by the instruction in line 3 inside the "while"-loop (Figure 3). With proper data structures such as integer counters storing the number of observed neighbors of a vertex, the application of the observation rules to a single vertex can be done in constant time. Post-order traversal of a rooted tree is clearly doable in linear time. □

4.2 Graphs of Bounded Treewidth

Our linear-time algorithm for graphs of bounded treewidth—assuming that the corresponding tree decomposition is given—uses basically the same strategy as the algorithms for DS [33, 34, 3, 6], that is, bottom-up dynamic programming from the leaves to the root. In the following, we demonstrate that there are two difficulties associated with OR2 that make PDS "harder" than DS and which cannot be solved by a simple modification of the algorithms for DS.

First, with OR2, there are more possibilities for a vertex to be observed. More precisely, the application of OR2 implies that there is a certain "observation dependency" between two vertices, that is, one vertex not in the power dominating set that becomes observed can make one of its neighbors observed. This observation dependency does not exist in DS. There, the domination status of one vertex that is not in the dominating set has no effect on the domination status of other vertices. Therefore, in order to describe this observation dependency in the dynamic programming, the three states defined in the algorithms for DS for the vertices in a bag, namely, "belonging to the dominating set," "already dominated at the current bag," and "not yet dominated at the current bag," are not sufficient. Second, only introducing further vertex states cannot settle the problem with the observation dependency. For example, assume that one defines the following additional state for the vertices in a bag: "already observed at the current bag by applying OR2 to one of its neighbors." Then, there could emerge a "cycle of observation dependencies" as follows: Considering a simple cycle as the input graph, one might assign the new state "observed due to OR2" to each vertex of the cycle. This is "locally correct" but globally it is false because the reasoning is done in a circular fashion without "global justification."

Our answer to these two difficulties is to define, in addition to the vertex states, three states for the edges in the subgraph induced by the vertices in a bag. In fact, these states give one of three possible *orientations* to an undirected edge $\{u, v\}$, orienting it from u to v, orienting it from v to u, or leaving it unoriented. These orientations express observation dependencies.

4.2.1 Valid Orientations of Undirected Graphs

In the following, we define the orientations.

Definition 4. An *orientation* of an undirected graph $G = (V, E)$ is a graph $D = (V, E_1 \dot{\cup} E_2)$ such that, for each $\{u, v\} \in E$, there is either a directed edge (u, v)

or (v, u) in E_1 or an undirected edge $\{u, v\}$ in E_2. The *indegree* of a vertex v in D, denoted by $d^-(v)$, is defined as $|\{(u, v) : (u, v) \in E_1\}|$ and the *outdegree* of v, denoted by $d^+(v)$, as $|\{(v, u) : (v, u) \in E_1\}|$. The subgraph $D[V']$ induced by $V' \subseteq V$ in D is called a *suborientation*.

Note that in the standard graph theory literature, an orientation of an undirected graph G is a directed graph D where there is exactly one of (u, v) and (v, u) in D for each edge $\{u, v\}$ in G. Here, we abuse the term orientation to denote a graph that results from orienting only a subset of edges. A directed edge (u, v) is called an *outgoing* edge for u and an *incoming* edge for v.

Definition 5. A *dependency path* in an orientation $D = (V, E_1 \cup E_2)$ is a subgraph of D consisting of a sequence of vertices and edges $v_1, e_1, v_2, e_2, \dots, e_{i-1}, v_i$ with $i \geq 3$, satisfying:

1. for all $1 \leq j, k \leq i$, $v_j = v_k \Leftrightarrow j = k$,
2. for all $1 \leq j \leq i - 1$, either $e_j = (v_j, v_{j+1}) \in E_1$ or $e_j = \{v_j, v_{j+1}\} \in E_2$,
3. and for all $1 \leq j \leq i - 1$, at least one of e_j and e_{j+1} is from E_1.

The vertices v_1 and v_i are called the *tail endpoint* and the *head endpoint* of the dependency path, respectively. A *dependency cycle* in an orientation is a dependency path with an edge between v_i and v_1 which can be undirected only if $(v_1, v_2) \in E_1$ and $(v_{i-1}, v_i) \in E_1$; otherwise, it is directed. A *directed path* is a dependency path with only directed edges.

Observe that a dependency path contains at least one directed edge.

Definition 6. A *valid orientation* $D = (V, E_1 \cup E_2)$ of an undirected graph $G = (V, E)$ is an orientation such that

1. $\forall v \in V$: $(d^-(v) \leq 1)$ and $(d^-(v) = 1 \Rightarrow d^+(v) \leq 1)$,
2. and there is no dependency cycle in D.

We call the set of vertices with $d^-(v) = 0$ the *origin* of D.

Note that each graph is a valid orientation of itself. See Figure 4 for an example of a valid orientation. Note that one can decide in $O(|E_1 \cup E_2|)$ time by depth-first search whether an orientation D is valid and, if so, find the origin of D. The following lemma is also easy to show.

Lemma 4. *Let $D = (V, E_1 \cup E_2)$ denote a valid orientation of an undirected graph G with origin $O \subseteq V$.*

1. *For each vertex $v \in V \setminus O$, there is exactly one directed path from the vertices in O to v.*
2. *Two directed paths from O to two vertices in $V \setminus O$ are vertex-disjoint with the possible exception of their tail endpoints in O.*

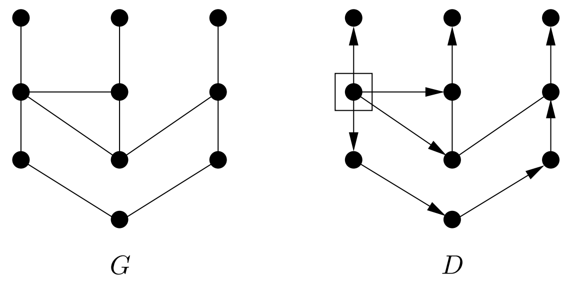

Figure 4: An example of a valid orientation D of an undirected graph G. The origin O of D contains only one vertex, marked with a rectangular box.

Proof. (1) First, we show that there is at least one directed path from a vertex in O to $v \notin O$. By Definition 6 we have $d^-(v) = 1$ with (u, v) denoting this edge. If $u \in O$, then edge (u, v) is the desired path; otherwise, $u \notin O$ and $d^-(u) = 1$. Again, we apply the above argument to u. Since graph G contains a finite number of vertices and there is no dependency cycle, there has to be a vertex $x \in O$ with a directed path from x to v. We denote this path as W.

The uniqueness of path W follows from the fact that, with the only exception of the tail in O, each vertex on this path has an indegree of exactly one.

(2) A directed path from a vertex in O to a vertex in $V \setminus O$ cannot pass through a vertex in O due to Definition 6. Two directed paths crossing at a vertex $u \in V \setminus O$ would imply $d^-(u) > 1$ or $d^+(u) > 1$ which is not allowed by Definition 6. This completes the proof. □

With the notations given above, we can introduce the following central orientation problem.

Valid Orientation with Minimum Origin (VOMO)
Input: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.
Question: Is there a vertex subset $O \subseteq V$ with $|O| \leq k$ such that G has a valid orientation with O as origin?

The following theorem shows that this orientation problem is a reformulation of PDS. This new problem formulation will be a decisive technical tool for our dynamic programming strategy to be described in Section 4.2.2.

Theorem 4. *An undirected graph G has a power dominating set P iff G has a valid orientation with P as origin.*

Proof. "$\Leftarrow$": Having a valid orientation D of G with $O \subseteq V$ as the origin, we show by contradiction that O is a power dominating set of G. See Figure 5 for an illustration. Suppose that O is not a power dominating set. Then, there is a vertex $v \in (V \setminus O)$ which is not observed by O. Since D is a valid orientation, there is a directed path p_1 from a vertex in O to v (Lemma 4). Assume without loss of generality that v is the first unobserved vertex on p_1. Let u denote the predecessor of v on p_1. Vertex u cannot be in O; otherwise v would be observed

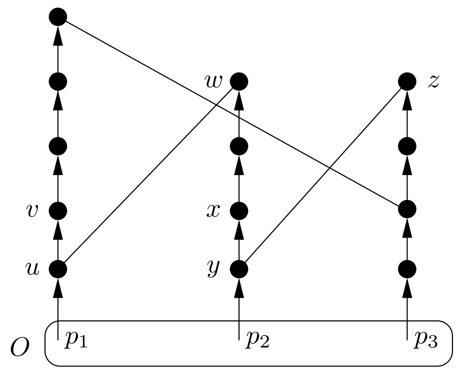

Figure 5: An example illustrating the proof of "$\Leftarrow$" of Theorem 4. Here, we assume that a dependency cycle is already built using three directed paths.

by OR1. Since u is observed and OR2 cannot be applied to u to observe v, there is another neighbor of u, denoted by w, which is not observed by O. Moreover, there is also a directed path p_2 in D from a vertex in O to w. Due to Lemma 4, p_1 and p_2 should be vertex-disjoint with the possible exception of their tail endpoints in O. Let x denote the first unobserved vertex on p_2. By applying the above argument now to x, vertex y which is x's predecessor on p_2 has an unobserved neighbor $z \neq x$ and there is a directed path p_3 from a vertex in O to z. Paths p_1, p_2, and p_3 are pairwise vertex-disjoint ignoring their tail endpoints in O. We apply this argument further to the first unobserved vertex on p_3 and so on. Since V is finite, we will encounter a vertex on one of the already considered paths $p_1, p_2, p_3, \ldots$ and, then, there exists a dependency cycle in D. This is a contradiction to the fact that D is a valid orientation.
"$\Rightarrow$": We prove this direction by describing a process which, for a given undirected graph $G = (V, E)$ with a power dominating set P, constructs a valid orientation D of G with P as origin.

Let $V' := V$ and $V_o := \emptyset$. The process simulates the applications of OR1 and OR2 and moves the vertices from V' to V_o. The sets V' and V_o store the unobserved and the observed vertices, respectively. For each vertex $v \in V \setminus P$, the process directs exactly one of the edges incident to v. An integer c counting the rounds of moves is initialized to zero and will be increased by one for each vertex moving from V' to V_o. In addition, the process records the number of the round when vertex v is moved from V' to V_o in an integer variable t_v; the value of t_v is set to the current value of c when moving v.

At the beginning, all vertices from P are moved from V' to V_o in an arbitrary order. Note that the count c increases continuously and variables t_v are set for each moved vertex v. Then, the process considers vertices $v \in V'$ which satisfy the following condition:

($\star$) $v \in N(u)$ for the vertex $u \in P$ which has the minimum t_u among all vertices $w \in P$ with $V' \cap N(w) \neq \emptyset$.

The process moves vertex v from V' to V_o and edge (u, v) is inserted into E_1. This operation is executed for all vertices in $V' \cap N(P)$. Note that now all applications of OR1 are simulated and V_o contains the vertices observed by OR1.

After that, if V' is not empty, then all vertices in V' have to be observed by OR2. Since P is a power dominating set, as long as V' is non-empty the process can always find a vertex $v \in V'$ which satisfies the following condition:

($\star\star$) $v \in N(u)$ where $u \in V_o$ with $(N[u] \setminus \{v\}) \subseteq V_o$.

Then, edge (u, v) is added to E_1 and $V' := V' \setminus \{v\}$ and $V_o := V_o \cup \{v\}$. After repeatedly applying this operation to each vertex in V', all edges in E which have no corresponding directed edges in E_1 are added to E_2 and $D := (V, E_1 \cup E_2)$. It is obvious that this process terminates after $|V|$ rounds because P is a power dominating set of G.

In the following, we prove that D is a valid orientation with P as origin.

It is clear that D is an orientation. In the process described above, we can also observe that $d^-(v) = 0$ for all $v \in P$ and a vertex is removed from V' only if it has received an incoming edge in E_1. Hence, we have $P = \{v \in V : d^-(v) = 0\}$. The first condition in the definition of valid orientations (Definition 6) is clearly satisfied by D: The vertices in P have no incoming edge and, because of ($\star$), every vertex $u \in N(P)$ has exactly one incoming edge from a vertex $v \in N(u) \cap P$ which has the minimum t_v among all vertices in $N(u) \cap P$. A vertex $u \in V \setminus (P \cup N(P))$ is moved from V' to V_o immediately after one of u's incident edges is directed to u. Thus, for all vertices $v \in V$, it holds $d^-(v) \leq 1$. Moreover, if a vertex u with $d^-(u) = 1$ has an outgoing edge (u, v), then $u \notin P$, vertex v is observed by applying OR2 to u, and $v \notin P \cup N(P)$. Since, by ($\star\star$), the process adds edge (u, v) to E_1 for a vertex $v \in V \setminus (P \cup N(P))$ only if $(N[u] \setminus \{v\}) \subseteq V_o$, the edge (u, v) is the only directed edge outgoing from u. Therefore, we obtain that $d^+(v) \leq 1$ for all vertices $v \in V$ with $d^-(v) = 1$. It remains to show that there is no dependency cycle in D.

Suppose that there is a dependency cycle in D and we order the vertices of this cycle $v_0, v_1, v_2, \ldots, v_i$ with $i > 2$ and $v_0 = v_i$ according to their appearance on the cycle. To simplify the presentation, we assume that $v_j = v_{j \bmod i}$.

We show that no vertex from P can be on this cycle: Suppose that this dependency cycle contains a vertex of P, that is, $v_j \in P$ with $0 \leq j < i$. The edge between v_{j-1} and v_j has to be undirected, since the process directs no edge to a vertex in P. Obviously, the edge between v_{j-2} and v_{j-1} has to be a directed edge, $(v_{j-2}, v_{j-1}) \in E_1$. Since a vertex from P has no incoming edge, we have $v_{j-1} \notin P$. As described above, the process firstly moves vertices in P from V' to V_o, then vertices in $N(P)$, and then vertices in $V \setminus (P \cup N(P))$. Since $v_{j-1} \in N(v_j) \subseteq N(P)$ and $(v_{j-2}, v_{j-1}) \in E_1$, it can be deduced that $v_{j-2} \in P$. Because of ($\star$) we can also infer $t_{v_{j-2}} < t_{v_j}$. Then, we apply the above argument for v_j again to v_{j-2} and can conclude that $v_{j-4} \in P$ and $t_{v_{j-4}} < t_{v_{j-2}}$. By applying the same argument to v_{j-4} and further vertices, we would have $t_{v_j} < t_{v_j}$ and, thus, the dependency cycle cannot contain vertices from P.

Hence, the dependency cycle only consists of vertices from $V \setminus P$ and the directed edges on this cycle represent the applications of OR2. If the edge between v_j and v_{j+1} is a directed edge, then $t_{v_{j+1}} > t_{v_j}$: The process gives $t_{v_{j+1}}$ a value greater than t_{v_j} after adding (v_j, v_{j+1}) to E_1. If the edge between v_j and v_{j+1} is an undirected edge, then $(v_{j+1}, v_{j+2}) \in E_1$ (the third point in Definition 5). Because of $(\star\star)$, v_{j+2} can be moved from V' to V_o and edge (v_{j+1}, v_{j+2}) can be added to E_1 only if all vertices in $N[v_{j+1}] \setminus \{v_{j+2}\}$ are already moved from V' to V_o. Thus, we know $t_{v_j} < t_{v_{j+2}}$. By applying this argument again and again to all directed and undirected edges in a cyclic fashion, we will get $t_{v_j} > t_{v_j}$. Hence, there is no dependency cycle and D is a valid orientation with P as the origin. □

With VOMO and Theorem 4, we have an alternative formulation of PDS. The advantage of VOMO is that, by orienting undirected edges, a dependency cycle, which corresponds to a cycle of observation dependencies in the power domination context, can be easily detected in the dynamic programming on the tree decomposition.

4.2.2 Dynamic Programming on Tree Decompositions

In what follows, we describe a dynamic programming procedure solving VOMO on graphs of bounded treewidth. As a consequence of Theorem 4, we thus also arrive at a solution for PDS.

Given an undirected, connected graph $G = (V, E)$ with $V := \{v_1, v_2, \ldots, v_n\}$ and a nice tree decomposition $\langle \{X_i : i \in I\}, T \rangle$ of G with treewidth k, let T_i denote the subtree of T rooted at node i and let G_i denote the subgraph of G induced by the vertices in the bags of T_i, that is, $G_i := G[\bigcup_{j \in V(T_i)} X_j]$. Furthermore, we use Y_i to denote $(\bigcup_{j \in V(T_i)} X_j) \setminus X_i$.

Definition of states: During a bottom-up process our dynamic programming algorithm computes for every bag X_i the possible valid orientations of the subgraph G_i and stores the sizes of the origins of the valid orientations. Herein, the valid orientations of G_i are characterized by the *bag states*. A bag state s of a bag X_i is a combination of the states for all ordered pairs of vertices in X_i, the states of all vertices in X_i, and the states of the edges in $G[X_i]$. Next, we will define the states for vertex pairs, for vertices, and for edges, respectively. Herein, we use $s(uv)$, $s(v)$, and $s(e)$ to denote the state of an ordered pair of vertices $u \in X_i$ and $v \in X_i$, the state of a vertex $v \in X_i$, and the state of an edge $e \in E(G[X_i])$.

The decisive point in the dynamic programming is to detect the dependency cycles in all possible orientations D_i for G_i when reaching bag X_i. The dependency cycles inside suborientations $D_i[X_i]$ can be detected by exhaustive search in $D_i[X_i]$ which consists of at most k vertices. However, for each pair of vertices u and v in X_i with $u \neq v$, the detection of dependency cycles passing through some vertices in Y_i is more involved. Such cycles consists of several dependency paths, some lying in $D_i[X_i]$ and others lying completely in $D_i[Y_i]$.

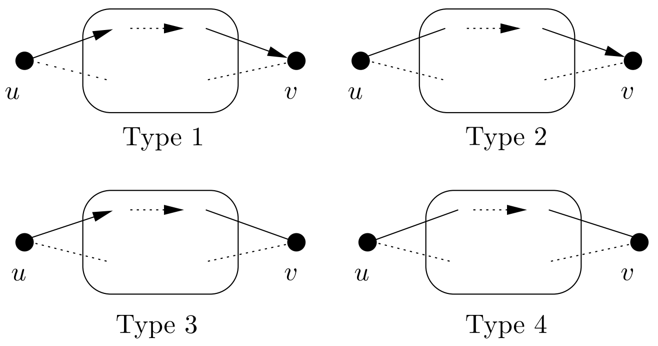

Figure 6: The four dependency path types from Definition 7.

By storing information about all dependency paths between all pairs of vertices $u, v \in X_i$ that, with the exception of u and v, only pass through vertices in Y_i, we will be able to detect all corresponding dependency cycles by exhaustive search in $O(k!)$ time. More specifically, we define for each pair of vertices in X_i several states reflecting the information about such dependency paths. Herein, let $V(p)$ be the set of the vertices on a dependency path p. We use p_{uv} to denote a dependency path with u and v as the tail and the head endpoint, respectively. The path p_{uv} is called a dependency path *from u to v*.

Definition 7. There are four types of dependency paths p_{uv} from u to v in an orientation D:

- Type 1: The first edge (between u and its successor on p_{uv}) as well as the last edge (between v and its predecessor on p_{uv}) of p_{uv} are directed edges.
- Type 2: The first edge of p_{uv} is a directed edge but the last edge is undirected.
- Type 3: The last edge of p_{uv} is a directed edge but the first edge is undirected.
- Type 4: Both the first edge and the last edge of p_{uv} are undirected edges.

In addition, the function g maps a dependency path p to one of the four types, that is, $g(p) \in \{\text{Type 1, Type 2, Type 3, Type 4}\}$.

Figure 6 illustrates the four path types. Observe that, if there are two dependency paths p_{uv} and p_{vu} in a valid orientation D, then $g(p_{uv}) \neq$ Type 1 and $g(p_{vu}) \neq$ Type 1, and at least one of p_{uv} and p_{vu} is Type 4 or one is Type 2 and the other is Type 3; otherwise, there is a dependency cycle in the suborientation $D[V(p_{uv}) \cup V(p_{vu})]$ which we call a "path type conflict." The states for an ordered vertex pair uv with $u \neq v$ in X_i are defined according to the possible types of dependency paths from u to v in $D_i[Y_i \cup \{u, v\}]$: There are 16 states for an ordered vertex pair uv according to the 16 possible subsets

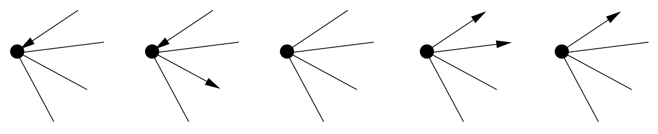

Figure 7: The five states of a vertex v in a bag.

of {Type 1, Type 2, Type 3, Type 4}. Such a state means that there is at least one dependency path between the vertex pair of each path type contained in the subset. With these vertex pair states, we can simply detect a dependency cycle at a bag X_i: check, for every two vertices u and v in X_i, whether there exists a path type conflict between $s(uv)$ and $s(vu)$, or whether there is a dependency path from u to v (or from v to u) in $D_i[X_i]$ whose type together with one type in $s(vu)$ (or in $s(uv)$) builds a path type conflict.

Next, we define five vertex states $s(v)$ for every vertex v in a bag X_i (see Figure 7 for an illustration):

- $s(v) = 1$: there is exactly one directed edge from a vertex in Y_i to v and no directed edge from v to vertices in Y_i;
- $s(v) = 2$: there is exactly one directed edge from a vertex in Y_i to v and exactly one directed edge from v to a vertex in Y_i;
- $s(v) = 3$: there is no directed edge between v and the vertices in Y_i;
- $s(v) = 4$: there are at least two directed edges from v to the vertices in Y_i and there is no directed edge from the vertices in Y_i to v;
- $s(v) = 5$: there is exactly one directed edge from v to a vertex in Y_i and no directed edge from the vertices in Y_i to v.

Note that, by the definition of valid orientations, a vertex in a valid orientation cannot have more than one incoming edge and a vertex with $N^-(v) = 1$ cannot have more than one outgoing edge. Hence, the above list covers all possible cases that need to be considered.

Furthermore, we define three edge states $s(e)$ for the edges $e = \{u, v\}$ in $G[X_i]$:

- $s(e) = uv$: edge e is directed from u to v,
- $s(e) = vu$: edge e is directed from v to u, and
- $s(e) = \perp$: edge e is undirected.

As a consequence, for a bag X_i with $|X_i| \leq k$, we have at most $16^{k(k-1)} \cdot 5^k \cdot 3^{k(k-1)/2}$ bag states. In the following, we say that a valid orientation D is *under the restriction of a bag state s of the bag X_i* if D satisfies the following conditions:

- the orientations in D of the edges in $G[X_i]$ coincide with their states given by s,
- for each ordered vertex pair uv with $u \in X_i$ and $v \in X_i$, the types of the dependency paths from u to v in $D[Y_i \cup \{u, v\}]$ coincide with $s(uv)$, and
- for each vertex $v \in X_i$, the orientations of the edges between v and the vertices in Y_i coincide with $s(v)$.

In the bottom-up dynamic programming we use a mapping A_i for each bag X_i which stores, for each bag state s, the minimum size of the origins of all possible valid orientations of G_i under the restriction of s. Due to the following easy-to-prove lemma, in the computation of the values for A_i we do not count vertices v for the size of an origin with $d^-(v) = 0$ and $d^+(v) \leq 1$.

Lemma 5. *For a connected, undirected graph $G = (V, E)$ with $|V| > 2$, there is always a valid orientation with a minimum origin $O \subseteq V$ such that each vertex $v \in O$ has in G at least two neighbors from $V \setminus O$ which are not neighbors of other vertices in O.*

Proof. The claim follows directly from Lemma 2 and the equivalence between PDS and VOMO in Theorem 4. □

Now, we have defined the states for a bag in a nice tree decomposition and proceed with the description of our dynamic programming for VOMO, which consists of an "initialization" and an "update phase." In the following, we use valid$(G[X_i], s_i)$ to denote the procedure deciding whether the edge states $s_i(e)$ of the edges e in $G[X_i]$ form a valid orientation of $G[X_i]$. Since $G[X_i]$ contains at most k vertices, valid$(G[X_i], s_i)$ needs at most $O(k!)$ time. Procedure valid$(G[X_i], s_i)$ returns true if s_i implies a valid orientation of $G[X_i]$; otherwise, it returns false.

Initialization: For each leaf node i with bag X_i of the tree decomposition T, we initialize A_i as follows. For each bag state s_i, we set $A_i(s_i) := +\infty$ if

$$(\exists v \in X_i : s_i(v) \neq 3) \vee (\exists uv : u \in X_i \wedge v \in X_i \wedge s_i(uv) \neq \emptyset) \vee (\text{valid}(G[X_i], s_i) = \text{false});$$

and, otherwise,

$$\begin{aligned} A_i(s_i) := &|\{v \in X_i \; : \; \exists e = \{v, u\} \in E(G[X_i]),\, e' = \{v, w\} \in E(G[X_i]) \text{ with} \\ & u \neq w \wedge s_i(e) = vu \wedge s_i(e') = vw\}|. \end{aligned}$$

By this initialization step, we make sure that only those bag states are taken into consideration where the edge states form a valid orientation. Also, since $Y_i = \emptyset$, $s(v)$ for each $v \in X_i$ should be assigned the vertex state 3, that is, there is no directed edge between v and vertices in Y_i, and $s(uv)$ for each ordered vertex pair $u \in X_i$ and $v \in X_i$ should be assigned the vertex pair state $\emptyset$, that is, there is no dependency path from u to v in $D_i[Y_i \cup \{u, v\}]$. Note that, due to Definition 6 and Lemma 5, as vertices in the origin we count only the vertices in X_i which are the tails of at least two directed edges in the valid orientation formed by the edge states in s_i.

Update phase: After the initialization, we visit the bags of the tree decomposition bottom-up from the leaves to the root, determining the corresponding mappings A_i in each step. For each bag state s_i of a bag X_i, the following needs to be done:

1. Check whether the edge states given by s_i form a valid orientation of $G[X_i]$.
2. Compute the set S^{s_i} of "with s_i compatible bag states" s_j (or "with s_i compatible bag state pairs" s_j and s_l for join nodes) of a child bag X_j (or two child bags X_j and X_l for join nodes). The formal definition of compatible bag states (and compatible bag state pairs) will be given individually for forget, insert, and join nodes, respectively (see the Appendix).
3. For a node i with a child node j, if G_i contains more edges than G_j, then, for each compatible bag state s_j (or each compatible bag state pair) in S^{s_i}, check whether a valid orientation of G_j under the restriction of s_j can be extended to a valid orientation for G_i under the restriction of s_i, that is, whether a valid orientation for G_j under the restriction of s_j combined with the orientations of the newly introduced edges in G_i which are given by s_i results in a valid orientation of G_i. For a node i with two children, the test whether the combination of two valid orientations is still a valid orientation of G_i can be carried out in a similar way.
4. Based on the mappings A_j for all compatible bag states (or bag state pairs) in S^{s_i}, evaluate $A_i(s_i)$.

The details of the dynamic programming are technical but use more or less standard methods. Hence, we defer these details into the Appendix.

When reaching the root of T, we can determine the minimum size of the origin of a valid orientation of G in the mapping A of the root node. Altogether, we thus arrive at the following central result.

Theorem 5. *For an n-vertex graph with given width-k tree decomposition,* Valid Orientation with Minimum Origin *can be solved in $O(c^{k^2} \cdot n)$ time for a constant c.*

Proof. The correctness of the algorithm follows directly from the description (also see Appendix). Concerning the running time, the most time-consuming part of the algorithm is the determination of the mapping A_i for a join node, where, for each bag state, we have to examine all possible bag state pairs, one from each child bag. As described above, there are less than $16^{k^2} \cdot 5^k \cdot 3^{k^2}$ bag states and, for each bag state s, there can be at most $(16^{k^2} \cdot 5^k \cdot 3^{k^2})^2$ bag state pairs which are compatible to s. For each compatible bag state pair (s_j, s_l), the dominating factor for the running time is to determine whether two valid orientations of G_j and G_l under the restrictions of s_j and s_l, respectively, can be combined into a valid orientation of G_i under the restriction of s_i. This can be done in $O(k!)$ time (see Appendix). In summary, this results in a worst-case running time of $O((16^{k^2} \cdot 5^k \cdot 3^{k^2})^3 \cdot k!)$ for a join node. □

Together with Theorem 4, we obtain our main result.

Theorem 6. *For an n-vertex graph with given width-k tree decomposition,* Power Dominating Set *can be solved in $O(c^{k^2} \cdot n)$ time for a constant c.*

5 Conclusion

Besides improving the exponential worst-case complexity of our dynamic programming algorithm in Section 4, there are several avenues for future work.

- Table 1 has several empty entries concerning the complexity of Power Dominating Set (PDS) in graph classes which we did not address here but have been addressed for Dominating Set (DS) [28]. In particular, is there a "significant" difference between the complexities of DS and PDS (also see the discussion in Section 1 and the very recent work of Aazami and Stilp [1])?
- Which graph classes where PDS is NP-complete allow for better than factor-$\Theta(\log n)$ polynomial-time approximation algorithms?
- How do fixed-parameter tractability results for DS in planar graphs [3, 4, 5] transfer to PDS?
- Are there nontrivial polynomial-time data reduction rules for PDS similar to those we have for DS [5]?
- Are there further "distance from triviality" parameterizations [20] that make PDS algorithmically feasible?

This paper particularly aims at stimulating a comparative study concerning the computational complexities of DS and PDS. So far, we are not aware of a graph class where DS is polynomial-time solvable and PDS is NP-complete or vice versa. In other words, the point is to better understand which role the second, non-local observation rule of PDS plays from an algorithmic viewpoint. We believe that PDS offers a promising field of research not only because of its practical importance but also because it allows for a compact problem formulation within which one can nicely study effects of non-locality by comparing the "local" DS with the "non-local" PDS problem. To understand this phenomenon in greater depth is a challenge for future research.

Acknowledgement. We are grateful to two anonymous referees of *Algorithmica* for their constructive feedbacks. In particular, one of the referees pointed out how to simplify the proofs of Lemma 2 and Theorem 1.

References

[1] A. Aazami and M. D. Stilp. Approximation algorithms and hardness for domination with propagation. In *Proc. 10th APPROX/11th RANDOM*, LNCS. Springer, 2007. To appear. 3, 22

[2] C. Adjih, P. Jacquet, and L. Viennot. Computing connected dominating sets with multipoint relays. *Ad Hoc & Sensor Wireless Networks*, 1(1–2):27–39, 2005. 2

[3] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for Dominating Set and related problems on planar graphs. *Algorithmica*, 33(4):461–493, 2002. 3, 12, 22

[4] J. Alber, H. Fan, M. R. Fellows, H. Fernau, R. Niedermeier, F. Rosamond, and U. Stege. A refined search tree technique for Dominating Set on planar graphs. *Journal of Computer and System Sciences*, 71(4):385–405, 2005. 22

[5] J. Alber, M. R. Fellows, and R. Niedermeier. Polynomial time data reduction for Dominating Set. *Journal of the ACM*, 51(3):363–384, 2004. 22

[6] J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proc. 5th LATIN*, volume 2286 of *LNCS*, pages 613–628. Springer, 2002. 12

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation — Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 1999. 2, 4

[8] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998. 5

[9] H. L. Bodlaender. Treewidth: Characterizations, applications, and computations. In *Proc. 32nd WG*, volume 4271 of *LNCS*, pages 1–14. Springer, 2006. 5

[10] K. S. Booth and J. H. Johnson. Dominating sets in chordal graphs. *SIAM Journal on Computing*, 11:191–199, 1982. 7

[11] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: a Survey.* SIAM Monographs on Discrete Mathematics and Applications, 1999. 6, 8

[12] D. J. Brueni and L. S. Heath. The PMU placement problem. *SIAM Journal on Discrete Mathematics*, 19(3):744–761, 2005. 2

[13] E. D. Demaine and M. Hajiaghayi. Bidimensionality: New connections between FPT algorithms and PTASs. In *Proc. 16th SODA*, pages 590–601. ACM/SIAM, 2005. 3

[14] A. K. Dewdney. Fast Turing reductions between problems in NP: chapter 4; reductions between NP-complete problems. Technical report, Department of Computer Science, University of Western Ontario, Canada, 1981. 7

[15] M. Dorfling and M. A. Henning. A note on power domination in grid graphs. *Discrete Applied Mathematics*, 154(6):1023–1027, 2006. 2

[16] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 2, 4, 7

[17] U. Feige. A threshold of $\ln n$ for approximating Set Cover. *Journal of the ACM*, 45(4):634–652, 1998. 2, 7

[18] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006. 2, 4

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. 4, 7

[20] J. Guo, F. Hüffner, and R. Niedermeier. A structural view on parameterizing problems: distance from triviality. In *Proc. 1st IWPEC*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004. 22

[21] T. W. Haynes, S. M. Hedetniemi, S. T. Hedetniemi, and M. A. Henning. Domination in graphs: applied to electric power networks. *SIAM Journal on Discrete Mathematics*, 15(4):519–529, 2002. 2, 3, 4, 6, 9

[22] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, editors. *Domination in Graphs: Advanced Topics*, volume 209 of *Pure and Applied Mathematics*. Marcel Dekker, 1998. 1

[23] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Pure and Applied Mathematics*. Marcel Dekker, 1998. 1

[24] T. W. Haynes and M. A. Henning. Domination in graphs. In J. L. Gross and J. Yellen, editors, *Handbook of Graph Theory*, pages 889–909. CRC Press, 2004. 1

[25] J. M. Keil. The complexity of domination problems in circle graphs. *Discrete Applied Mathematics*, 36:25–34, 1992. 7

[26] T. Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994. 5, 6

[27] J. Kneis, D. Mölle, S. Richter, and P. Rossmanith. Parameterized power domination complexity. *Information Processing Letters*, 98(4):145–149, 2006. 2, 3, 4, 7, 9

[28] D. Kratsch. Algorithms. In T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, editors, *Domination in Graphs: Advanced Topics*, pages 191–231. Marcel Dekker, 1998. 8, 9, 22

[29] C. S. Liao and D. T. Lee. Power dominating problem in graphs. In *Proc. 11th COCOON*, volume 3595 of *LNCS*, pages 818–828. Springer, 2005. 2, 3, 7, 8, 9

[30] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006. 2, 4

[31] D. Raible. *Algorithms and Complexity Results for Power Domination in Networks* (in German). Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2005. 8

[32] B. A. Reed. Algorithmic aspects of tree width. In B. A. Reed and C. L. Sales, editors, *Recent Advances in Algorithms and Combinatorics*, pages 85–107. Springer, 2003. 5

[33] J. A. Telle and A. Proskurowski. Practical algorithms on partial k-trees with an application to domination-like problems. In *Proc. 3rd WADS*, volume 709 of *LNCS*, pages 610–621. Springer, 1993. 12

[34] J. A. Telle and A. Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics.*, 10(4):529–550, 1997. 12

[35] V. V. Vazirani. *Approximation Algorithms.* Springer, 2001. 2, 4

Appendix:

Details of the dynamic programming.

Forget Nodes: Suppose that node i is a forget node with child node j, that is, $X_i := \{x_{i_1}, \ldots, x_{i_{n_i}}\}$ and $X_j := \{x_{i_1}, \ldots, x_{i_{n_i}}, x\}$. Since G_i and G_j contain the same set of edges, each valid orientation of G_j is also a valid orientation of G_i.

For each bag state s_i of node i, we compute the set S^{s_i} containing the bag states s_j of node j that are compatible with s_i. Note that, if there exists an edge between a vertex $v \in X_i$ and the vertex x, then the orientation of edge $\{v, x\}$ can result in a difference between $s_i(v)$ and $s_j(v)$. For example, a vertex with $s_j(v) = 4$ having an edge $\{v, x\}$ with $s_j(\{v, x\}) = vx$ should have $s_i(v) = 5$. The reason is that now vertex v has two outgoing edges to vertices in Y_i, the new one being (v, x). Herein, we say that s_j is compatible with s_i *with respect to a vertex* $v \in X_i$ if one of the following is true:

- $s_i(v) = s_j(v) \wedge (\{x, v\} \notin E \vee ((e = \{x, v\} \in E) \wedge (s_j(e) = \perp)))$,
- $s_j(v) = 3 \wedge s_i(v) = 1 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = xv))$,
- $s_j(v) = 1 \wedge s_i(v) = 2 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = vx))$,
- $s_j(v) = 5 \wedge s_i(v) = 2 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = xv))$,
- $s_j(v) = 4 \wedge s_i(v) = 4 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = vx))$,
- $s_j(v) = 5 \wedge s_i(v) = 4 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = vx))$,
- $s_j(v) = 3 \wedge s_i(v) = 5 \wedge ((e = \{x, v\} \in E) \wedge (s_j(e) = vx))$.

To define the compatibility of s_j with respect to an ordered vertex pair, we introduce a so-called *concatenation* function of path types. This partial function maps two given path types to a path type. If the path resulting by identifying the head endpoint of a path of the first path type and the tail endpoint of a path of the second path type is a dependency path, then this function is defined and returns the type of the resulting path; otherwise, it is not defined. For example, if the first path type is Type 1 and the second is Type 2, then the function returns Type 2; if the first path type is Type 2 and the second is Type 3, the resulting path is not a dependency path and the concatenation function is not defined.

We say that s_j is compatible with s_i *with respect to an ordered vertex pair* uv with $u \in X_i$ and $v \in X_i$ if $s_i(uv)$ extends $s_j(uv)$ by concatenating the path types in $s_j(ux)$ and the path types in $s_j(xv)$ and by adding the type of the possible dependency path from u to v formed by edges $\{u, x\}$ and $\{v, x\}$. For example, if edges $e = \{u, x\}$ and $e' = \{v, x\}$ are in E and $s_j(e) = ux$ and $s_j(e') = \perp$, then we have a Type-2-path from u to v passing through x. Hence, s_j is compatible with s_i with respect to uv if $s_j(uv) = \{\text{Type 1, Type 4}\}$, $s_j(ux) = \emptyset$, $s_j(xv) = \emptyset$, and $s_i(uv) = \{\text{Type 1, Type 2, Type 4}\}$.

The bag states $s_j \in S^{s_i}$ that are compatible with s_i have to satisfy the following conditions:

- for each $e \in E(G[X_i])$, $s_i(e) = s_j(e)$,
- s_j is compatible with s_i with respect to each $v \in X_i$, and
- s_j is compatible with s_i with respect to each ordered vertex pair uv with $u \in X_i$ and $v \in X_i$.

Then, evaluate the mapping A_i of X_i as follows: For each bag state s_i, set

$$A_i(s_i) := \min_{s_j \in S^{s_i}} \{A_j(s_j)\}.$$

Note that, if $S^{s_i} = \emptyset$ for an s_i, then $A_i(s_i) = +\infty$. The computation of $A_i(s_i)$ is correct since G_i and G_j have the same set of edges and, thus, each valid orientation of G_i is also a valid orientation of G_j.

Introduce Nodes: Suppose that node i is an introduce node with child node j, that is $X_i := \{x_{j_1}, \ldots, x_{j_{n_j}}, x\}$ and $X_j := \{x_{j_1}, \ldots, x_{j_{n_j}}\}$.

Procedure valid$(G[X_i], s_i)$ can decide whether the edge states contained in s_i form a valid orientation of $G[X_i]$ in $O(k!)$ time. For each bag state s_i of node i, we compute the set S^{s_i} containing the with s_i compatible bag states s_j of node j which satisfy

$$(\forall e \in E(G[X_j]) : s_i(e) = s_j(e)) \wedge (\forall v : s_i(v) = s_j(v)) \wedge (\forall uv : s_i(uv) = s_j(uv)),$$

where $u, v \in X_j$.

Note that the introduction of a new vertex x does not change the number of directed edges between a vertex $v \in X_j$ and the vertices in Y_i; thus, $s_i(v) = s_j(v)$. The types of the dependency paths remain the same for each ordered vertex pair uv with $u \in X_j$ and $v \in X_j$, that is, $s_i(uv) = s_j(uv)$. Moreover, since there is no edge between the new vertex $x \in X_i$ and the vertices in Y_i (due to the consistency property of tree decompositions), we set $S^{s_i} := \emptyset$ for bag states s_i where $s_i(x) \neq 3$ or where there is a vertex $v \in X_j$ with $s_i(vx) \neq \emptyset$ or $s_i(xv) \neq \emptyset$.

The next task is to decide whether a valid orientation of G_j under the restriction of a bag state $s_j \in S^{s_i}$ can be extended to a valid orientation of G_i under the restriction of s_i. Herein, note that G_i has one more vertex x than G_j and, hence, we have to take into account the states of the edges incident to x in $G[X_i]$. Observe that the conditions for a valid orientation can be violated by a vertex in $N(x) \cap X_i$ or by a dependency cycle passing through vertex x. First, for each vertex v in $N(x) \cap X_i$, we can easily find out the indegree and outdegree of v in the orientation of G_j under the restriction of s_j, based on the information stored in $s_j(v)$ and $s_j(e)$ for all edges e in $E(G_j)$ incident to v. Together with the orientation of edge $e = \{v, x\}$ given by $s_i(e)$, the indegree and outdegree of v in the orientation of G_i under the restriction of s_i can be easily determined. Hence, to check whether a vertex $v \in N(x) \cap X_i$ violates the vertex-degree conditions of a valid orientation, that is, $d^-(v) \leq 1$ and $d^-(v) = 1 \Rightarrow d^+(v) \leq 1$, can be done in $O(k^2)$ time because there can be at most $O(k^2)$ edges between vertices in X_j. Second, if there is a dependency cycle passing through x in the orientation of G_i under the restriction of s_i, then it consists of some consecutive dependency paths between some pairs of the vertices in X_j and two edges incident on x. By accessing the information about the dependency path types stored in $s_j(vu)$ for each ordered vertex pair with $u, v \in X_j$, an exhaustive search running in $O(k!)$ time can easily decide whether there are such dependency cycles. Altogether, we can in $O(k!)$ time decide whether a valid orientation of G_j under the restriction of a bag state $s_j \in S^{s_i}$ can be extended to a valid orientation of G_i under the restriction of s_i. If the valid orientation of G_j under the restriction of $s_j \in S^{s_i}$ cannot be extended to a valid orientation of G_i under the restriction of s_i, then we remove s_j from S^{s_i}.

Finally, we compute the mapping $A_i(s_i)$ for s_i based on the mappings $A_j(s_j)$ for all $s_j \in S^{s_i}$. By orienting the edges which are incident to the new vertex x according to the edge states given by s_i, vertex x could have more than one outgoing edge. Moreover, the outdegrees of vertices $v \in X_i$ which are adjacent to x and have outdegree one in the valid orientation of G_j under the restriction of s_j could also become two. As a consequence, we should add x and those of x's neighbors which now have outdegree two into the origin of the valid orientation of G_i under the restriction of s_i. Therefore, we introduce in the following two functions B and f to cope with these possible new origin vertices.

The mapping A_i for X_i is computed as follows: For each bag state s_i, set

$$A_i(s_i) := \min_{s_j \in S^{s_i}} \{A_j(s_j) + f_{s_i}(x) + |B_{s_i}(s_j)|\},$$

where $f_{s_i}(x) = 1$ if there exist at least two edges $e = \{u, x\}$ and $e' = \{v, x\}$ in $G[X_i]$ with $s_i(e) = xu$ and $s_i(e') = xv$; otherwise, $f_{s_i}(x) = 0$. The function $B_{s_i}(s_j)$ returns the set of vertices $u \in (N(x) \cap X_j)$ which satisfy one of the following two conditions:

- $s_j(u) = 5$ and $s_i(\{u, x\}) = ux$ or
- $s_j(u) = 3$, $s_i(\{u, x\}) = ux$, and $s_i(\{u, v\}) = uv$ for an edge $\{u, v\} \in E(G[X_i])$ with $v \neq x$.

Roughly speaking, $f_{s_i}(x) = 1$ covers the case that x has to be added to the origin and the set returned by $B_{s_i}(s_j)$ contains the vertices which are in the origin of the valid orientation of G_i but not in the origin of the valid orientation of G_j. Note that, if $S^{s_i} = \emptyset$ for a s_i, then $A_i(s_i) = +\infty$.

Join Nodes: Suppose that i is a join node with child nodes j and l, that is $X_i := \{x_{i_1}, \ldots, x_{i_{n_i}}\}$ and $X_i = X_j = X_l$. The considerations for join nodes are almost the same as for forget and introduce nodes. There are two points to note here.

Let S^{s_i} be the set of the bag state pairs (s_j, s_l) that are compatible with s_i. Note that, due to the consistency property of tree decompositions, $Y_j \cap Y_l = \emptyset$. Because $Y_i = Y_j \cup Y_l$, the indegree (outdegree, respectively) of v in $G[Y_i \cup \{v\}]$ is equal to the sum of the indegrees (outdegrees, respectively) of v in $G[Y_j \cup \{v\}]$ and in $G[Y_l \cup \{v\}]$. For example, if $s_j(v) = 1$ and $s_l(v) = 5$, then v has an incoming edge from a vertex in Y_j and an outgoing edge to a vertex in Y_l and, thus, $s_i(v) = 2$. Therefore, we say that s_i is compatible with s_j and s_l *with respect to vertex* $v \in X_i$ if one of the following is true:

- $s_i(v) = 1 \wedge ((s_j(v) = 1 \wedge s_l(v) = 3) \vee (s_j(v) = 3 \wedge s_l(v) = 1))$,
- $s_i(v) = 2 \wedge ((s_j(v) = 2 \wedge s_l(v) = 3) \vee (s_j(v) = 3 \wedge s_l(v) = 2) \vee (s_j(v) = 1 \wedge s_l(v) = 5) \vee (s_j(v) = 5 \wedge s_l(v) = 1))$,
- $s_i(v) = 3 \wedge s_j(v) = 3 \wedge s_l(v) = 3$,
- $s_i(v) = 4 \wedge ((s_j(v) = 4 \wedge s_l(v) = 3) \vee (s_j(v) = 3 \wedge s_l(v) = 4) \vee (s_j(v) = 5 \wedge s_l(v) = 5))$,
- $s_i(v) = 5 \wedge ((s_j(v) = 5 \wedge s_l(v) = 3) \vee (s_j(v) = 3 \wedge s_l(v) = 5))$.

With the fact that $Y_j \cap Y_l = \emptyset$, a dependency path from vertex $u \in X_i$ to vertex $v \in X_i$ passing through vertices in Y_j is then, with the exceptions of u and v, vertex-disjoint with any dependency path from u to v passing through vertices in Y_l. Thus, the set of dependency paths from u to v passing through vertices in Y_i is then the union of the set of dependency paths from u to v passing through vertices in Y_j and the set of dependency paths from u to v passing through vertices in Y_l. Therefore, we say that s_i is compatible with s_j and s_l *with respect to an ordered vertex pair* uv with $u \in X_i$ and $v \in X_i$ if $s_i(uv)$ is the union of $s_j(uv)$ and $s_l(uv)$.

In summary, the bag state pairs $(s_j, s_l) \in S^{s_i}$ that are compitable with s_i satisfy the following conditions:

- for each $e \in E(G[X_i])$, $s_i(e) = s_j(e) = s_l(e)$,
- s_i is compatible with s_j and s_l with respect to each $v \in X_i$, and
- s_i is compatible with s_j and s_l with respect to each ordered vertex pair uv with $u \in X_i$ and $v \in X_i$.

The next task is to check whether the combination of the two valid orientations of graph G_j and G_l which are under the restrictions of s_j and s_l, respectively, is a valid orientation of G_i under the restriction of s_i. This can be done in almost the same way as for the introduce nodes. The running time is $O(k!)$.

Finally, in order to compute the mapping $A_i(s_i)$ for a bag state s_i, observe that, by combining two valid orientations of G_j and G_l, there could emerge new origin vertices in the valid orientation of G_i. For example, a vertex $v \in X_i$ has in each of the two valid orientations of G_j and G_l only one outgoing edge, one to a vertex in Y_j and one to a vertex in Y_l. Since $Y_i = Y_j \cup Y_l$, vertex v has now two outgoing edges in the valid orientation of G_i resulting from the combination of the valid orientations of G_j and G_l. The function B introduced below counts such new origin vertices. Moreover, there could be vertices in X_i which are in the origins of both valid orientations of G_j and G_l. Hence, by combining the two valid orientations and summing up the size of the origins of the two valid orientations, there could be some vertices counted twice. In order to avoid such double counting we introduce a function C in the computation of $A_i(s_i)$.

For each bag state s_i, the determination of A_i is done as follows:

$$A_i(s_i) := \min_{(s_j,s_l)\in S^{s_i}} \{A_j(s_j) + A_l(s_l) + |B_{s_i}(s_j,s_l)| - |C_{s_i}(s_j,s_l)|\},$$

where function $B_{s_i}(s_j,s_l)$ returns the set of new origin vertices $v \in X_i$, that is, $s_j(v) = 5$ and $s_l(v) = 5$ and there is no edge $e = \{u,v\} \in E(G[X_i])$ with $s_i(e) = uv$, and function $C_{s_i}(s_j,s_l)$ contains the vertices $v \in X_i$ which satisfy one of the following conditions:

- there are at least two edges $e = \{u,v\}$, $e' = \{v,w\}$ in $E(G[X_i])$ with $s_i(e) = vu$ and $s_i(e') = vw$,
- $s_j(v) = 5$ and $s_l(v) = 5$ and there is exactly one edge $e = \{u,v\}$ in $E(G[X_i])$ with $s_i(e) = vu$, or
- $s_j(v) = 4$ and $s_l(v) = 4$.