Karlsruhe Institute of Technology

Algorithms for k-Power Dominating Set

Bachelor Thesis of

Tobias Hoch

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	Dr. rer. nat. Torsten Ueckerdt
Advisor:	Max Göttlicher

28.06.2024 – 28.10.2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 28.10.2024

. . . . . .

(Tobias Hoch)

The Power Dominating Set (PDS) problem is a graph covering problem with significant applications in energy informatics. The goal is to identify the smallest subset $S \subseteq V$ such that every vertex in the graph is monitored according to specific rules. In this paper, we extend the traditional PDS problem by introducing the k-Power Dominating Set (k-PDS), a generalized variant that considers an additional parameter k. We begin by examining existing reduction rules for PDS, adapting them to the k-PDS context where applicable, and deriving new reduction rules unique to this variant. Additionally, we propose an efficient algorithm that solves both PDS and k-PDS on cactus graphs with linear time complexity.

Abstract

Das Power Dominating Set (PDS)-Problem ist ein „graph covering“ Problem mit wichtigen Anwendungen in der Energieinformatik. Ziel ist es, die kleinste Teilmenge $S \subseteq V$ zu identifizieren, sodass jeder Knoten im Graphen gemäß spezifischer Regeln beobachtet wird. In dieser Arbeit erweitern wir das traditionelle PDS-Problem durch die Einführung des k-Power Dominating Set (k-PDS), einer verallgemeinerten Variante, die einen zusätzlichen Parameter k berücksichtigt. Wir beginnen mit der Untersuchung bestehender Reduktionsregeln für PDS, passen diese an den k-PDS-Kontext an, wenn möglich, und führen neue, Reduktionsregeln für diese Variante ein. Zusätzlich stellen wir einen effizienten Algorithmus vor, der sowohl PDS als auch k-PDS auf Kaktusgraphen in Linearzeit löst.

Zusammenfassung

Contents

1 Introduction

Efficient monitoring of networks, such as electrical power grids, requires strategic selection of a minimal set of vertices. The Power Dominating Set (PDS) problem formalizes this task, seeking the smallest subset of vertices to ensure full coverage of the network through propagation mechanisms. This problem is used in energy informatics to optimize the placement of costly measurement devices, called phonon measurement units (PMUs), in power grids, ensuring comprehensive system observation with minimal resource use.

Motivated by the practical application of the PDS problem, we explore a generalization known as the K-Power Dominating Set (k-PDS) problem first introduced by Chang et al. [CDMR12]. The k-PDS problem introduces an additional parameter k, which alters one propagation rule and potentially allows for a more flexible monitoring process. This generalization opens up new possibilities for theoretical exploration and may offer insights into more complex monitoring scenarios.

Although the k-PDS problem is primarily studied from a theoretical perspective, understanding its properties and developing efficient algorithms can provide valuable insights and solutions to upcoming challenges in the future.

The decision problem associated with both PDS and k-PDS is known to be NP-complete, which highlights the computational complexity of finding exact solutions for general instances. Our goal is to find reduction rules for k-PDS giving us an equivalent instance, for which certain vertices are pre-selected or forbidden to select, significantly reducing solving time.

In this paper, we first examine existing reduction rules introduced by Bläsius and Göttlicher [BG23] for PDS and explore their applicability to k-PDS. We then derive new reduction rules unique to k-PDS, expanding the theoretical framework of the problem. Additionally, we present a linear-time algorithm for solving both PDS and k-PDS on cactus graphs. Although an algorithm for solving PDS on graphs with a given width-k tree decomposition has already been introduced by Guo et al. [GNR08], our approach to k-PDS. Finally, we evaluate our results on a range of existing electrical networks and other benchmark instances.

2 Preliminaries

2.1 Graph Theory

Definition 2.1 (Graphs and Neighborhoods ([BG23], [Die17], pp. 2, 5))**:** *Let $G = (V, E)$ be an undirected graph with vertices V and edges E. For $v \in V$, let $N(v) = \{u \in V \mid uv \in E\}$ be the open neighborhood of v. Similarly, $N[v] = N(v) \cup \{v\}$ is the closed neighborhood of v. Given a set $S \subseteq V$, we denote by $N(S)$ and $N[S]$ the union of all open and closed neighborhoods of the vertices in S.*

Definition 2.2 (Orientation of a graph [[GNR08] p. 188)**:** *] An orientation of an undirected graph $G = (V, E)$ is a graph $D = (V, E_1 \cup E_2)$ 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 $|\{(v, u) : (v, u) \in E_1\}|$.*

Definition 2.3 (Connected Graph ([Die17], p. 10))**:** *A graph G is called connected if it is non-empty and any two of its vertices are linked by a path in G.*

Definition 2.4 (Biconnected Component (Block) ([Die17], p. 60))**:** *A biconnected component (or block) of a graph $G = (V, E)$ is a maximal subgraph of G that remains connected upon the removal of any single vertex.*

Definition 2.5 (Cut Vertex ([Die17], p. 11))**:** *A cut vertex (or articulation point) is a vertex whose removal disconnects the graph into at least two connected components.*

Definition 2.6 (Cactus Graph ([Wes01], p. 199))**:** *A cactus graph is a connected graph in which any two cycles have at most one vertex in common. Equivalently, a cactus graph is characterized by the property that every edge belongs to at most one cycle.*

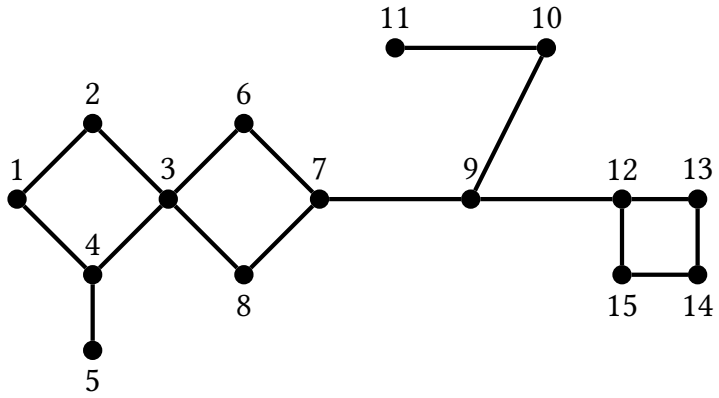


Figure 2.1: Example of a cactus graph

Definition 2.7 (Block-Cut Tree ([Wes01], p. 201))**:** *A block-cut tree is a tree that represents the structure of a graph in terms of its biconnected components (blocks) and cut vertices. In a block-cut tree, each node represents either a biconnected component (block node) or a cut vertex (cut node) of the original graph. There is an edge between a block node and a cut vertex node in the block-cut tree if and only if the corresponding cut vertex is part of the corresponding biconnected component.*

The structure of a block-cut tree is bipartite, meaning edges alternate between block nodes and cut nodes.
Note that in the case of a cactus graph, each block is either a cycle or a path consisting of two vertices.

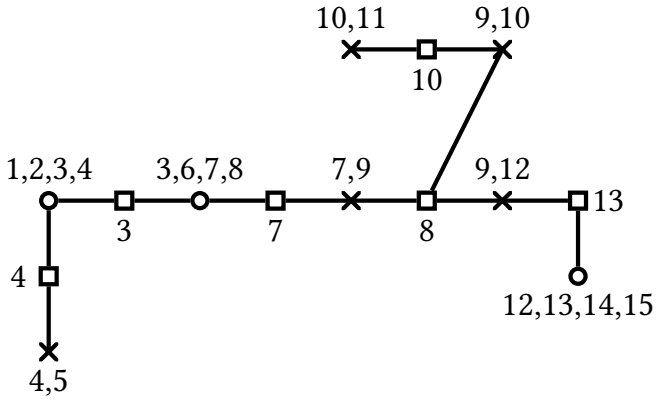


Figure 2.2: The Block-Cut Tree corresponding to the cactus graph in Figure 2.1. Square nodes represent cut vertices, circular nodes represent cycles, and crossed nodes represent paths of two vertices in the original cactus graph.

2.2 Power Dominating Set

Definition 2.8 (Power Dominating Set [BG23])**:** *For a given graph G, the problem Power Dominating Set (PDS) is to find a minimum vertex set $S \subseteq V$ of selected vertices such that all vertices of the graph are observed. Such a set is called a power dominating set. The size of a minimum power dominating set of a given graph G is called the power dominating number $\gamma_P(G)$. Whether a vertex is observed is determined by the following rules, which are applied iteratively. In the following, we often refer to them as observation rules one and two or short, OR1 and OR2. Note that for the second rule, vertices can be marked as propagating, i.e., the input of PDS is not just a graph but a graph together with a set of propagating vertices.*

***Domination rule**. A vertex is observed if it is in the closed neighborhood of a selected vertex.*
***Propagation rule**. Let $u \in V$ be a propagating vertex. If u is observed and $v \in N(u)$ is the only neighbor of u that is not yet observed, then v becomes observed. If the propagation rule is applied to an observed vertex u, we say it propagates its observation status.*

The following k-Power Dominating Set problem extends simple Power Dominating Set by allowing an observed vertex to propagate its observation to all its unobserved neighbors, provided it has at most k such neighbors. In contrast, the simple PDS case corresponds to $k = 1$, where an observed vertex can only propagate if it has exactly one unobserved neighbor.

Definition 2.9 (k-Power Dominating Set): *Given a graph $G = (V, E)$ and a positive integer $k \in \mathbb{N}_0$, a set of vertices $S \subseteq V$ is called a k-Power Dominating Set (k-PDS) if all vertices in G can be observed by iteratively applying the following two observation rules.*

***Domination Rule**. Initially, every vertex in the closed neighborhood $N[S]$ of S is observed.*
***Propagation Rule**. Any vertex $u \in V$ that is already observed and has **at most** k **unobserved neighbors** propagate its observation status to all its unobserved neighbors.*
The objective is to find a minimum-sized set S for the given graph G so all vertices of G are observed.

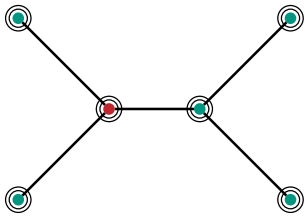


Figure 2.3: Example of a k-PDS instance, with $k = 2$. The propagation potential π of each vertex is indicated by the number of cycles around each node. Red vertices are selected, green vertices are observed but not pre-selected.

The following Potential Power Dominating Set problem generalizes the k-PDS problem by allowing each vertex u to have a distinct *propagation potential*, given by $\pi(u)$. In contrast to k-PDS, where all vertices share the same propagation potential k, PPDS allows for a vertex-specific propagation potential.

Definition 2.10 (Potential Power Dominating Set): *Given a graph $G = (V, E)$ and a function $\pi : V \to \mathbb{N}_0$ that assigns to each vertex $u \in V$ a propagation potential $\pi(u)$, a non-negative integer representing the maximum number of unobserved neighbors u can have while still being able to propagate its observation status. A set of vertices $S \subseteq V$ is called a Potential Power Dominating Set (PPDS) if all vertices in G can be observed by iteratively applying the following two observation rules:*

***Domination Rule**. Initially, every vertex in the closed neighborhood $N[S]$ of S is observed.*
***Propagation Rule**. Any vertex $u \in V$ that is observed and has **at most** $\pi(u)$ **unobserved neighbors** can propagate its observation status to all its unobserved neighbors.*
The objective is to find a minimum-sized set S for the given graph G such that all vertices of G are observed.

In the following sections, we frequently work with the *extension* variant of the previously introduced problems. The k-PDS-Extension problem is defined as follows: the input consists of a graph G, a parameter k, and two additional sets of vertices, $X \subseteq V$ (the *pre-selected* vertices) and $Y \subseteq V$ (the *excluded* vertices). The remaining vertices, $V \setminus X \setminus Y$, are referred to as *undecided*. The goal of k-PDS-Extension is to find a minimum-sized set $S \subseteq V$ that includes all vertices in X, excludes all vertices in Y, and ensures that every vertex in the graph is observed. The problems PDS-Extension and PPDS-Extension are defined analogously. To distinguish this variant from the standard problem, we refer to the original k-PDS problem as simple-k-PDS.

As previously noted, the PPDS problem is a generalization of the k-PDS problem. Since we intend to use PPDS-Extension to address k-PDS instances, it is important to demonstrate that any PPDS-Extension instance, where no vertex has a propagation potential greater than k, can be efficiently transformed into a k-PDS instance.

Lemma 2.11: *A PPDS instance with each vertex having a propagation potential of at most k can be transformed into an equivalent k-PDS instance.*

Proof. Let $G = (V, E)$ be a PPDS instance. If a vertex $v \in V$ is observed and has at most $\pi(v)$ unobserved neighbors, it propagates its observation status to all its neighbors. Now, consider adding an (unobserved) leaf $w \notin V$ to G with the edge $\{w, v\}$. The vertex v can only propagate its observation to its neighbors if it has at most $\pi(v) - 1$ other unobserved neighbors than w.

Furthermore, let S be a minimal solution to the PPDS problem. If $w \in S$, an equivalent solution $S' = (S \setminus \{w\}) \cup \{v\}$ can always be constructed. Thus, attaching a leaf to a vertex v and increasing v's propagation potential by one generates an equivalent instance.

As a result, by attaching $k - \pi(u)$ additional leaf vertices to each vertex $u \in V$ and setting its propagation potential to k, we obtain an equivalent k-PDS instance. These added leaves ensure that all vertices in the transformed graph have a uniform propagation potential of k, preserving the structure of the original problem. ∎

Definition 2.12 (Observation Neighborhood)**:** *Let $U \subseteq V$ be a set of vertices. The observation neighborhood of U is the set of vertices that is observed after adding U to the set of already selected vertices. The observation neighborhood for a single vertex v is defined analogously, where $U = \{v\}$.*

3 k-PDS-Extension Reduction Rules

In this section, we propose a set of reduction rules designed to minimize the solving time of the k-PDS-Extension problem for arbitrary instances. Although the decision problem for k-PDS (and thus k-PDS-Extension) has been proven to be NP-complete by Chang et al. [CDMR12], significant gains in solving efficiency can be made by reducing the size of the instance, preselecting certain vertices, and restricting others from being chosen. As demonstrated in Section 5, these reduction rules have led to substantial improvements in computational performance, demonstrating their practical value. The reduction rules we present below generate an instance that is equivalent to the original in terms of solving k-PDS-Extension.

It is important to note that our reduction rules produce an instance of the PPDS-Extension Problem. However, as demonstrated in the Preleminaries 2.2, PPDS-Extension instances can be transformed into k-PDS-Extension instances.

We further examine the reduction rules for PDS-Extension, proposed by Bläsius and Göttlicher [BG23] and assess their applicability to K-PDS-Extension. Additionally, we introduce a new reduction rule that works particularly well with the new possibilities of the propagation rule in k-PDS-Extension.

3.1 Applicability of existing local PDS-Extension Reduction Rules to PPDS-Extension

We begin by examining local reduction rules, which are rules that make decisions based on specific substructures within a graph. Recall that each vertex can be in one of several states: pre-selected, excluded, or undecided, as is typical for instances of the extension problem. Additionally, every vertex u has a corresponding propagation potential $\pi(u)$. In the original PDS problem, a vertex is either propagating or non-propagating, which corresponds to having a propagation potential of either 0 or 1. For the corresponding proofs, refer to the end of each subsection.

In [BG23], Bläsius and Göttlicher observe that leaves are almost never part of an optimal solution. The first reduction rule (Deg1a, Deg1b) they propose is as follows: if a leaf is attached to an undecided propagating vertex, the leaf is deleted and the undecided vertex is made non-propagating. Alternatively, if the leaf is attached to an undecided non-propagating vertex, the leaf is deleted and the undecided vertex is pre-selected.

Following this principle, we formulate the corresponding reduction rule for k-PDS:

Reduction CollapseLeaves **(CL).** Let u be a leaf attached to a non-excluded vertex v. If v has a non-zero propagation potential ($\pi(v) > 0$), decrease its propagation potential by one and remove the leaf vertex u. If $\pi(v) = 0$, preselect v, i.e., set $X \leftarrow X \cup \{v\}$, and delete u.

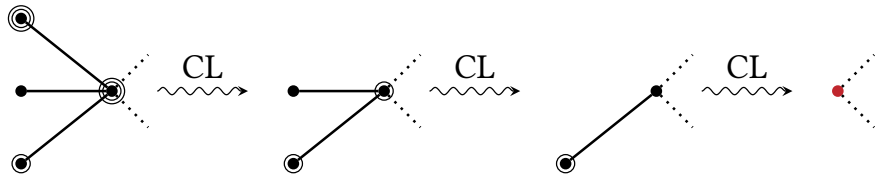


Figure 3.1: Sequence showing the application of the CollapseLeaves reduction rule. The amount of black circles around a vertex denote their propagation potential, and a black vertex indicates, that it is unobserved, a red vertex indicates that it is selected.

Note two important points: First, the propagation potential of leaf vertices does not affect rule applicability. Second, even after a vertex v is selected, this reduction rule continues to check for any leaves attached to v and removes them, further reducing the number of vertices in the graph.

Next, Bläsius and Göttlicher identify several trivial cases where a vertex must be selected, which are directly applicable to our problem. Of these, we highlight two, as the remaining cases are edge cases addressed by our newly introduced reduction rule later in this section.

Reduction Isolated (Isol)[BG23]. Let v be an undecided isolated vertex. Then pre-select v.

Reduction Observed Non-Propagating (ObsNP)[BG23]. Let v be an observed, excluded vertex with propagation potential of zero. Then delete v.

Furthermore, they make a key observation: the number of edges in the graph can be safely reduced. Vertices observed through the propagation rule can instead be observed via the domination rule. To achieve this, all observed vertices are directly connected to a selected vertex, after which any remaining edges to observed neighbors are removed.

Reduction Observed Edge (ObsE).[BG23]. Let $\{v, w\} \in E$ be an edge with two observed but not pre-selected endpoints and let $x \in X$ be a pre-selected vertex. Then delete $\{v, w\}$ and instead insert edges $\{v, x\}$ and $\{w, x\}$.

Observe that, under this approach, any vertex with only observed neighbors effectively becomes a leaf. This rule pairs particularly well with the CollapseLeaves reduction rule, as leaves created by ObsE will subsequently be removed by CL.

Lemma 3.1: *Reduction CL is safe.*

Proof. Given a graph $G = (V, E)$, let $v \in V$ be a leaf node with neighbor $w \in V$. Let S be a minimal solution for the problem. If $v \in S$ and $w \notin Y$, we can construct an equivalent solution $S' = (S \setminus \{v\}) \cup \{w\}$. Therefore, we can conclude that there is always a minimal solution without v.

Observe that v, being a leaf, cannot be observed by any vertex other than its neighbor w. Thus, we consider the following two cases:

Case 1: w has a propagation potential greater than zero. In this case, w can propagate its observation status to v w is observed and its propagation potential is at least one higher than the amount of other unobserved neighbors.

Since a power dominating set requires all vertices to be observed, v must be observed by w in a valid solution. Therefore, it is safe to remove v and decrement the propagation potential of w by one.

Case 2: w has a propagation potential of zero. If w cannot propagate its observation to any other vertex, v can only be observed by w by the domination rule. Therefore, w must be selected.

In both cases, the safety of removing v from the solution is ensured, either by adjusting the propagation potential of w or by selecting w. ∎

Their proofs for the following lemmas 3.2, 3.3, 3.4 remain valid as they do not depend on the propagation rule. And thus also hold for PPDS-Extension.

Lemma 3.2: *Reduction **Isol** is safe.*

Proof. Isolated vertices cannot be observed by any other node so they must be selected. ∎

Lemma 3.3: *[BG23] Reduction **ObsNP** is safe.*

Proof. [BG23] Let v be an observed inactive non-zero-injection vertex as above. Such a vertex can never propagate and does not count toward the unobserved neighbors of any other vertex. The observation rules can thus not be applied to v and application of the propagation rule to any of the neighbors of v does not depend on v. It is thus safe to remove v. ∎

Lemma 3.4: *Reduction **ObsE** is safe.*

Proof. [BG23] After application of the reduction, both vertices have a selected neighbor and can thus be observed by the domination rule, so they remain observed. Their number of unobserved neighbors does not change and thus the propagation rule can be applied to them as before. ∎

3.2 Merge-Low-Degree Reduction Rule for PPDS-Extension

In this subsection, we propose a new reduction rule that aims to scale especially well with increasing propagation potential of vertices, improving efficiency in reducing PPDS-Extension instances. This is demonstrated in Section 5.3, were we evaluate the reduction rules on a range of practical instances.

Note that this reduction rule increases the propagation potential of certain vertices in the graph. Consequently, the reduction to a k-PDS-Extension instance, as outlined in the Preliminaries 2.2, is no longer applicable after applying this reduction rule to a PPDS-Extension instance.

The key idea is to identify subgraphs within the graph where, if one vertex in the subgraph is observed, every vertex in the subghraph is also observed by the propagation rule. This condition holds when each vertex v in a induced subgraph $G[V'], V' \subseteq V$ has an degree that is at most its propagation potential plus one.

When one vertex in $G[V']$ is observed from the outside, a cascading effect occurs in which the unobserved degree of every vertex in the subgraph is reduced by at least one. As a result, the propagation rule will observe all vertices in the subgraph, as well as their immediate neighbors $N(V')$. This allows us to replace the entire subgraph with a much smaller, equivalent gadget, in the original graph, leading to the following reduction rule:

Reduction Merge Low Degree (MLD). Given a graph $G = (V, E)$. Let $M = G[V'], V' \subseteq b$ be an induced subgraph of G with the property that $\forall v \in V' : d(v) \leq \pi(v) + 1$ and v is unobserved. Let $d_M(n)$ denote the number of edges from a vertex $n \in V \setminus V'$ to any vertex in the subgraph M, where $V' \subseteq V$ is the vertex set of M. Formally, $d_M(n) = |\{\{n, v\} \mid v \in V' \text{ and } \{n, v\} \in E\}|$.

Remove each vertex $v \in V'$ and its incident edges, replacing it with a gadget consisting of $\max d_M(n) \mid n \in V \setminus V'$ vertices. Let the vertices of the gadget form a set $M_{\_}$. Set the propagation potential of each vertex $v \in M_{\_}$ in the gadget to infinity, and connect each initial neighbor $n \in N(V')$ to $d_M(n)$ vertices in $M_{\_}$.

Additionally, if at least one neighbor $n \in N(V')$ is not excluded, exclude every vertex $v \in M_{\_}$ of the gadget. If $|N(V')| > 1$, $|N(V')| \neq 0$, or if all neighbors $n \in N(V')$ are excluded, add a path between the vertices in $M_{\_}$.

When we first apply the MLD reduction rule in an instance of k-PDS-extension, specifically, an instance of PPDS-extension where each vertex has a propagation potential of k, the rule effectively reduces the remaining decisions on to vertices with a degree greater than $k + 1$. This makes the rule especially effective for instances with large values of k.

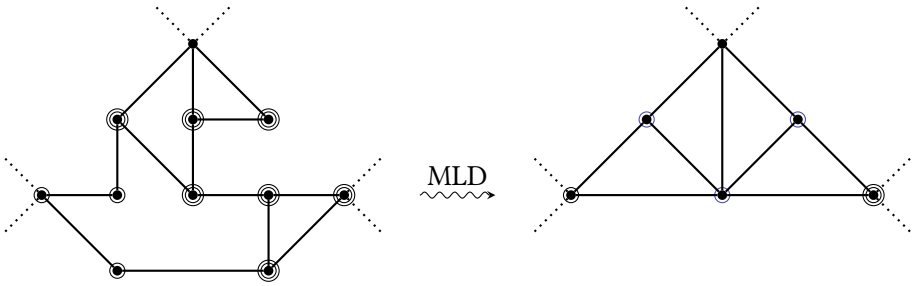


Figure 3.2: Example of the application of the Merge Low Degree reduction rule. Blue circles around vertices indicate a propagation potential of infinity.

Moreover, the MLD reduction rule integrates well with existing reduction rules. If a subgraph identified by the MLD rule has only one neighboring vertex, all vertices within gadget become leaves. By subsequently applying the CollapseLeaves (CL) reduction, these leaves are removed, and the neighboring vertex might be selected.

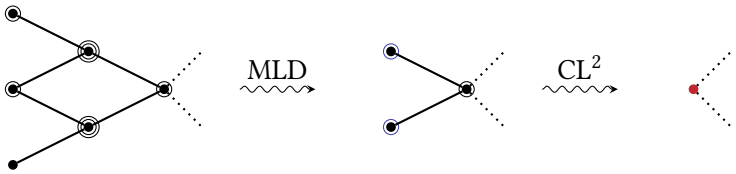


Figure 3.3: Example of the Merge Low Degree (MLD) reduction rule applied to a subgraph with a only single neighbor, followed by two applications of the Collapse Leaves (CL) reduction. The final result is a selected vertex marked in red.

Lemma 3.5: *Reduction **MLD** is safe.*

Proof. We begin by noting that the unobserved degree of the neighbors of the specified subgraph in the reduction rule remains unchanged. This is because we add exactly the same number of edges to unobserved vertices as existed before the reduction. Therefore, the result of the reduction is equivalent from the perspective of the neighbors of such a subgraph.

Since the subgraph $M = G[V']$ is connected and every vertex within the subgraph has a propagation potential less than or equal to its degree plus one, and none of the vertices are observed initially, it follows that once one vertex v is observed, either through a neighbor being selected, or observed through propagation, the unobserved degree v and thus of other vertices in the subgraph decreases by at least one. As a result, the propagation rule observes one vertex at a time until all vertices and its neighbors $N[V']$ are observed. Replacing this subgraph with the simplified gadget, which introduces a path between the vertices, preserves this property.

If at least one neighboring vertex is not excluded, we can safely exclude the vertices in M (or equivalently, M_-) because the *observation neighborhood*—the set of vertices that is observed if after adding a vertex to the selected vertices—of any neighbor vertex is a superset of the observation neighborhood of any vertex within the subgraph.

If $|N(V')| = 1$ and the single neighbor $n \in N(V')$ is not excluded, there is no need to add a path between the vertices in M_-. In this case, the vertices M_- become leaves with n as their only neighbor. If one leaf is observed by n, then all leaves in M_- are observed by n, thus observing all of M_-. Consequently, the MLD reduction is safe. ∎

3.3 Applicability of existing global PDS-Extension Reduction Rules to PPDS-Extension

Now, we turn to global reduction rules. Unlike the previous rules that focused on small structures within the graph, these rules involve making observations about the graph as a whole. Bläsius and Göttlicher introduced two highly effective global reduction rules for the Power Dominating Set (PDS) problem. These rules can be directly applied to the PPDS-Extension problem and are formulated as follows.

Reduction Domination (Dom)[BG23]. If the closed neighborhood of some undecided vertex w is contained in the observation neighborhood of some other undecided vertex v, exclude w.

The following reduction rule generalizes local reduction rules, such as the Collapse Leaves rule, which identify vertices that must be selected in a specific minimum solution. However, similar to the isolated vertices handled by the Isol reduction rule, there exist vertices that must be selected in every minimum solution. Such vertices can be identified if they are not part of the observation neighborhood of the set of all other undecided vertices. This leads to the following reduction rule:

Reduction Necessary Node (NecN)[BG23]. Let $v \in B$ be an undecided vertex. If the observation neighborhood of all undecided vertices except v does not contain all vertices in G, select v.

Lemma 3.6: *Reduction **Dom** is safe.*

Proof. [BG23] Because $N[w]$ is contained in the observation neighborhood of v, w can never observe any vertices that are not observed when selecting v. Thus, a power dominating set containing w can never be smaller than one containing v, and v can safely be excluded. ∎

Lemma 3.7: *Reduction **NecN** is safe.*

Proof. [BG23] Selecting more vertices cannot make fewer vertices observed. Thus, if the observation neighborhood of all blank vertices except v does not contain the whole graph, there exists no power dominating set that does not contain v. ∎

4 k-PDS Cactus Algorithm

In this section, we present an algorithm that efficiently solves the k-PDS problem on cactus graphs with linear-time complexity. Due to the propagation rule, the observation of a single vertex can have a cascading effect throughout the graph, influencing the observation status of potentially all vertices. To address the various possibilities of how observation can spread, we use a dynamic programming approach that evaluates multiple propagation scenarios and selects the optimal combination at the final vertex. The central idea of the algorithm is to simulate the observation process by orienting edges to indicate the direction of observation for each vertex. This orientation helps to track and manage how the observation propagates across the graph, ultimately enabling the selection of the most efficient solution. Below, we illustrate an example of such an orientation for a graph with $k = 1$.

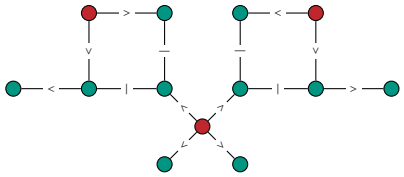


Figure 4.1: Example of a k-PDS with $k = 1$, where $<$ and $>$ denote the direction from where a vertex is observed, and $|$ that there is no observation rule needed on this edge.

Such an approach was first proposed by Guo et al. [GNR08] in the context of the Power Dominating Set (PDS) problem. The objective was to find a *valid orientation* of the graph that corresponds to a minimum-sized set of vertices capable of observing the entire graph. We build upon this idea and adapt it to the context of the k-PDS problem.

In our approach, each edge in the graph can either be undirected ($|$) or directed from one vertex u to another v, represented as either $<$ or $>$. The orientation of edges is limited by the observation rules of the k-PDS problem. According to the domination rule, a selected vertex observes all its neighbors, which is modeled by directing all edges incident to a selected vertex as outgoing. In the case where two adjacent vertices are selected, the edge between them is undirected. For the propagation rule, if a vertex is observed, it can propagate its observation to at most k unobserved neighbors, provided it has no additional unobserved neighbors beyond this limit. To simulate this behavior, a vertex is allowed up to k outgoing edges as long as it has at least one incoming edge, indicating that the vertex itself is observed by another vertex. Additionally, any number of undirected edges may connect an observed vertex to other vertices, indicating these vertices are observed by other parts of the graph.

With this understanding, we want to define rules for a *valid orientation* on a graph. However, based on the previous observations alone, an invalid orientation can still occur within cycles. This prohibited structure is referred to as a ”dependency cycle” in [GNR08].

In simple terms, a cycle is a dependency cycle if its edges are consistently oriented in one direction (either entirely clockwise or counterclockwise) or are undirected, with the additional condition that no two consecutive edges can be undirected. The formal Definition is provided below.

Definition 4.1 (Dependency Cycle)**:** *Let C_n be a cycle with vertices $\{v_1, \ldots, v_n\}$ and edges e_j for $j \in \{1, \ldots, n\}$, where $e_j = \{v_j, v_{(j+1) \bmod n}\}$. We call such a cycle a dependency cycle if the following conditions are satisfied:*

1. *One of the following conditions must hold:*
   - *For all* $1 \leq j \leq n$*, the edge* e_j *is either directed from* v_j *to* $v_{(j+1) \bmod n}$ *or remains undirected.*
   - *For all* $1 \leq j \leq n$*, the edge* e_j *is either directed from* $v_{(j+1) \bmod n}$ *to* v_j *or remains undirected.*
2. *For each* $1 \leq j \leq n$*, at least one of the edges* e_j *or* $e_{(j+1) \bmod n}$ *must be directed.*

Recall that in a cactus graph an edge can only be part of one cycle. Thus it easy to check for dependency cycles in cactus graphs.

Various examples and counterexamples of dependency cycles (for $k > 0$) are shown below.

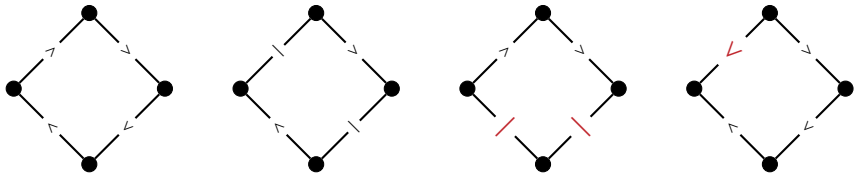


Figure 4.2: Examples (a, b) and counterexamples (c, d) of dependency cycles.

In Figure 4.2 (a), the conditions for applying the second observation rule are satisfied, with each vertex having an incoming edge and $\leq k$ outgoing edges. This suggests a valid orientation. Nevertheless, it is evident that this would not translate to a valid solution of k-PDS, since no vertex would be selected, e.g. the propagation has no clear origin.

With that we define the following rules, derived from the Definition of a valid orientation in [GNR08].

Definition 4.2 (Valid orientaion)**:** *A valid orientation* D *of an undirected graph* $G = (V, E)$*, with* $k \in \mathbb{N}_0$*, is an orientation such that*

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

The set of vertices with $d^-(v) = 0$ *is called the origin* O *of* D*, which corresponds to the set* S *in the k-PDS problem.*

With this concept, we introduce the following orientation problem, which forms the basis of the algorithm to solve k-PDS on cactus graphs.

Problem [Valid Orientation with Minimum Origin (VOMO) [GNR08]]
Input: An undirected graph $G = (V, E)$ and an integer $l \geq 0$.
Question: Is there a vertex subset $O \subseteq V$ with $|O| \leq l$ such that G has a valid orientation with O as origin?

The following theorem demonstrates that VOMO, combined with our Definition of valid orientations, provides a reformulation of the k-PDS problem.

Theorem 4.3: *A graph* G *has a k-power dominating set* S *if and only if* G *has a valid orientation with* S *as the origin.*

To prove the theorem, we make use of the lemma below regarding valid orientations. This observation was originally made by Guo et al. [GNR08]. It remains applicable to our Definition of valid orientations, as its proof relies solely on the condition that for all vertices not part of the origin, their in-degree is exactly one, i.e., $d^-(v) = 1$.

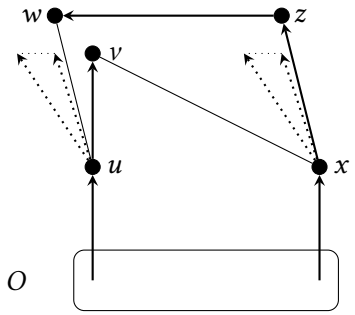


Figure 4.3: Example illustrating the first part of the proof (" $\Leftarrow$ " :) of Theorem 4.3. Where the vertices u, v, z, x, w form a dependency cycle. With $p_1 = (O, u, v)$ and $p_2 = (O, z, x, w)$

Lemma 4.4: *Let $D = (V, E_1 \cup E_2)$ denote a valid orientation of an undirected graph G with origin $O \subseteq V$. For each vertex $v \in V \setminus O$, there is exactly one directed path from the vertices in O to v.*

Now we can prove Theorem 4.3, following the original structure of [GNR08].

Proof. " $\Leftarrow$ " : Assume we have a valid orientation D of G with $O \subseteq V$ as the origin. We will prove by contradiction that O is a k-power dominating set of G.

Suppose, for the sake of contradiction, that O is not a k-power dominating set. Then, there must exist a vertex $v \in (V \setminus O)$ that is not observed by O. Since D is a valid orientation, there exists a directed path p_1 from some vertex $o_1 \in O$ to v (by Lemma 4.4). Without loss of generality, let v be the first unobserved vertex on p_1, and let u denote the predecessor of v on this path.

The vertex u cannot belong to O, as otherwise, v would be observed by the domination rule. Since u is observed but not propagating (otherwise it would have observed v), it follows that u must have at least $k + 1$ unobserved neighbors, including v. However, since D is a valid orientation, we know that $d^+(u) \leq k$, meaning that u must have at least one undirected edge connecting it to another unobserved neighbor, say w.

By Lemma 4.4, there is also a directed path p_2 from a vertex of the origin O to w. Let x denote the first unobserved vertex on this path. Note that x cannot belong to p_1, as v is the first unobserved vertex on p_1, and clearly, $x \neq v$. This vertex again, has at least one neighbor, connected by an undirected edge.

We can apply this reasoning iteratively to further paths. Since V is finite, we will eventually encounter a vertex that lies on an already considered path. This would imply the existence of a dependency cycle in D, which contradicts the assumption that D is a valid orientation.

For an illustration, where a dependency cycle is constructed by only examingin two such paths described above, see4.3.

" $\Rightarrow$ " : We now prove the reverse direction by describing a process that constructs a valid orientation D for G, given a k-power dominating set $S \subseteq V$.

Let $V' := V$ and $V_o := \emptyset$. The process simulates the application of the observation rules and moves vertices from V' to V_o. The sets V' and V_o represent the unobserved and observed vertices, respectively. For each vertex $v \in V \setminus S$, the process orients exactly one of the edges incident to v. We use an integer c to count the number of rounds, incrementing by one each time a vertex moves from V' to V_o. For each vertex v, let t_v denote the round when it is moved from V' to V_o.

Initially, all vertices in S are moved in an arbitrary order, from V' to V_o. Note that by each vertex moved from V' to V_o the count c increases, and t_v is set for each vertex moved. The next step is to move all the vertices that satisfy the following condition.
($\star$) $v \in N(u)$ for the vertex $u \in S$ which has the minimum t_u among all vertices $w \in S$ with $V' \cap N(w) \neq \emptyset$.

The process transitions vertex v from V' to V_o and inserts the edge (u, v) into E_1, where u is an observed vertex in $N(v)$. This operation is repeated for all vertices in $V' \cap N(S)$. At this stage, all applications of the domination rule have been simulated, and V_o now contains the vertices observed by the domination rule.

If V' is not empty after this step, the remaining vertices in V' must be observed through the propagation rule. Since S is a k-power dominating set, as long as V' is non-empty, the process can always identify a set of vertices $V \subseteq V'$ that satisfies the following condition:
($\star\star$) $V \subseteq N(u)$, where $u \in V_o$ and $(N[u] \setminus V) \subseteq V_o, |V| \leq k$.

Next, for each $v \in V$, the edges (u, v) are added to E_1, and the sets are updated as follows: $V' := V' \setminus V$ and $V_o := V_o \cup V$. It is important to note that all vertices $v \in V$ are assigned the same value of t_v. By repeatedly applying this operation to each vertex in V', all remaining edges in E that do not already have corresponding directed edges in E_1 are added to E_2. Thus, the orientation $D := (V, E_1 \cup E_2)$ is constructed. Clearly, this process terminates in at most $|V|$ rounds, as S is a k-power dominating set of G, and in each round, at least one vertex is added to V_o.

Now we prove that the resulting orientation D is a valid orientation (Definition 4.2). It is evident that D is indeed an orientation. From the process outlined above, we can observe that $d^-(v) = 0$ for every vertex $v \in S$, and a vertex is removed from V' only when it has received an incoming edge in E_1. Therefore, we have $S = \{v \in V : d^-(v) = 0\}$. The first condition in the Definition of valid orientations is thus satisfied by D: vertices in S do not have incoming edges, and by condition ($\star$), every vertex $u \in N(S)$ has precisely one incoming edge from a vertex $v \in N(u) \cap S$ that has the smallest t_v among all vertices in $N(u) \cap S$. A vertex $u \in V \setminus (S \cup N(S))$ is moved from V' to V_o immediately after one of its incident edges is directed to u. Thus, for all vertices $v \in V$, it holds that $d^-(v) \leq 1$. Furthermore, if a vertex u with $d^-(u) = 1$ has an outgoing edge (u, v), then $u \notin S$, and vertex v is observed by applying the propagation rule to u, and $v \notin S \cup N(S)$. Since, by condition ($\star\star$), the process adds the edge (u, v) to E_1 for a vertex $v \in V \setminus (S \cup N(S))$ only when $(N[u] \setminus V) \subseteq V_o$, (with $|V| \leq k$, and $v \in V$), u has at maximum k outgoing edges. Therefore, we conclude that $d^+(v) \leq k$ for all vertices $v \in V$ with $d^-(v) = 1$. It remains to demonstrate 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 $v_j = v_{j \mod i}$.

We show that no vertex from S can be on this cycle. Suppose this dependency cycle contains a vertex of S, i.e., $v_j \in S$ with $0 \leq j < i$. The edge between v_{j-1} and v_j must be undirected, as no edge is directed to a vertex in S. Evidently, the edge between v_{j-2} and v_{j-1} must be a directed edge, $(v_{j-2}, v_{j-1}) \in E_1$. Since a vertex from S has no incoming edges, we have $v_{j-1} \notin S$. As previously explained, the process initially moves vertices in S from V' to V_o, followed by vertices in $N(S)$, and then vertices in $V \setminus (S \cup N(S))$. Since $v_{j-1} \in N(v_j) \subseteq N(S)$ and $(v_{j-2}, v_{j-1}) \in E_1$, it can be inferred that $v_{j-2} \in S$. Due to condition ($\star$), we also know that $t_{v_{j-2}} < t_{v_j}$. Therefore, by applying the same argument to v_j again for v_{j-2}, we can conclude that $v_{j-4} \in S$, and $t_{v_{j-4}} < t_{v_{j-2}}$. By repeating this argument for subsequent vertices, we observe that $t_{v_j} < t_{v_j}$, leading to a contradiction, which means that the dependency cycle cannot contain any vertices from S.

Hence, the dependency cycle consists only of vertices from $V \setminus S$, and the directed edges on this cycle represent applications of the propagation rule. If the edge between v_j and v_{j+1} is directed, then $t_{v_{j+1}} > t_{v_j}$, as the process assigns $t_{v_{j+1}}$ a value greater than t_{v_j} after adding the edge (v_j, v_{j+1}) to E_1. If the edge between v_j and v_{j+1} is undirected, then $(v_{j+1}, v_{j+2}) \in E_1$ (based on the third condition in Definition 4.1).

Due to condition ($\star\star$), it follows that $t_{v_j} \leq t_{v_{j+2}}$. This is because when the edge (v_{j+1}, v_{j+2}) is added to E_1, the edges from v_{j+1} to all other neighbors of v_{j+1} in V' are also added to V_o. As a result, v_j is either included in this iteration or was already added to V_o in a previous round. Therefore, we have two possible cases:

- If $t_{v_j} = t_{v_{j+2}}$, then both (v_{j+1}, v_j) and (v_{j+1}, v_{j+2}) would be included in E_1, which would contradict the first condition of a dependency cycle (Definition 4.1).

- Otherwise, $t_{v_j} < t_{v_{j+2}}$. By iterating this argument over all directed and undirected edges in the cycle, we eventually conclude that $t_{v_j} < t_{v_j}$, which is a contradiction.

Thus, no dependency cycle can exist in D, and therefore D is a valid orientation with S as the origin. ∎

With VOMO and Theorem 4.3, we now have an alternative formulation of the k-PDS problem. This reformulation offers a significant advantage, as it allows us to leverage dynamic programming to efficiently check the first condition of Definition 4.2 at each step. Moreover, since we are working on cactus graphs—where each edge belongs to at most one cycle—we can detect potential dependency cycles at the critical edges where cycles close. This ensures that both conditions of the valid orientation are met as the algorithm progresses.

4.1 Algorithm for k-PDS on Cactus Graphs

In the following, we propose a dynamic programming approach to efficiently solve the VOMO problem, and thereby the k-PDS problem, on cactus graphs. Note that Hon et al. proposed a linear-time algorithm for power dominating set on block cactus graphs [HLPT07]. Here, we present an alternative approach that not only achieves similar efficiency but also generalizes to solve k-PDS on cactus graphs. The objective is to identify a valid orientation by assigning costs to each possible edge direction in a bottom-up manner, ultimately selecting the optimal combination at the final vertex. This cost function, serving as a central tool for the algorithm to solve the VOMO problem, reflects the number of vertices chosen as origins based on edge orientation and will be introduced in detail later in this section.

4.1.1 Initialization and Preparation

The algorithm begins by establishing an order in which the edges of the graph are processed. Recall that cactus graphs consist of two types of blocks: cycles and paths of two vertices. The algorithm leverages this structure as follows: first, a root vertex r is artificially added to the graph, connected by a single edge $e = (r, v)$, where $v \in V$. This augmentation results in a new graph G', which remains a cactus graph. The introduction of the root is essential for ensuring that edge orientations can be finalized systematically during the last step.

Next, the cactus graph G' is transformed into its corresponding block-cut tree, where blocks (either cycles or paths) and cut vertices are organized in a tree structure. In this tree, every vertex, except for the root, has exactly one parent.

This property allows us to process the graph in a bottom-up manner. We achieve this by employing a reverse topological ordering of the edges, starting from the deepest parts of the block-cut tree and progressing upwards. This layered approach ensures that each edge connecting to a vertex in a higher layer is examined only after all edges from vertices in lower layers have been fully processed.

A visual example illustrating these initial steps is provided below.

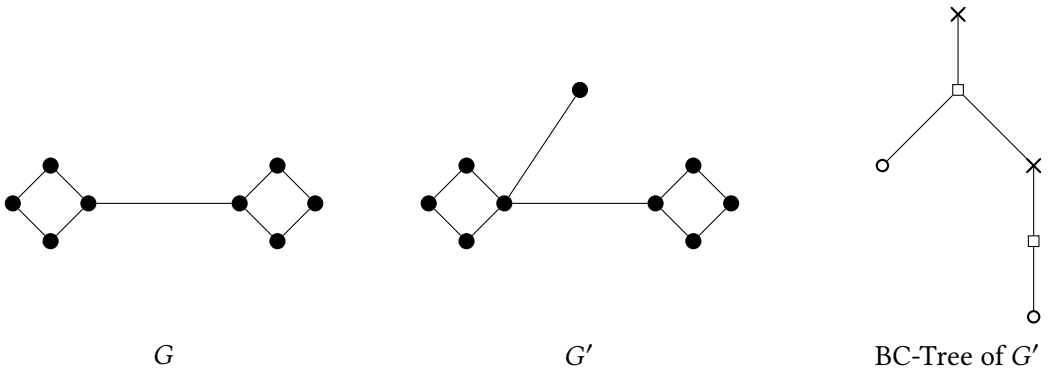


Figure 4.4: Illustration of the graph transformation process used by the algorithm. Starting with the initial graph G (left), the root vertex is added to form G' (center), and the corresponding block-cut tree of G' is constructed (right). In the BC-Tree, a cross node indicates a path block containing two initial vertices, a rectangular node represents a cut vertex, and a circular node stands for a cycle block.

The next step of the algorithm calculates the minimal costs associated with each valid orientation of the edges in the graph. The algorithm primarily operates on the block-cut tree to compute costs between components and cut vertices. However, for cycle components, it reverts to the original graph representation and processes each edge within the cycle in a clockwise manner. Details on how the costs are calculated will be discussed in greater depth later in this section.

4.1.2 Edge Orientation

From this point onward, we will refer to an edge as a *back edge* if it is oriented against the examination order. In the block-cut tree, this means a back edge is directed from a vertex at a higher layer toward its children. Similarly, in a cycle component with a set embedding, a back edge is one oriented counter-clockwise. In contrast, a *forward edge* follows the examination order: from a vertex to its parent in the block-cut tree or in a clockwise direction within a cycle component. We refer to an undirected edge as a *neutral edge*

We will later see that another advantage of the block-cut tree is that it simplifies the graph's structure. Instead of managing multiple edges connected to a cut vertex (as in the case where a cut vertex belongs to a cycle), the block-cut tree represents only a single edge leading to the cut vertex and then to another block. This abstraction enables the algorithm to calculate the costs for exactly one edge at a time.

Nevertheless, to avoid incorrect cost calculations, it is necessary to introduce additional orientation options in the block-cut tree. Specifically, when a cycle is connected to a cut vertex, an edge in the block-cut tree must simulate the interaction of two actual edges from the original cycle.

An example for a cycle on a lower layer connected to a cut vertex on a higher layer is shown below.

Figure 4.5: Example, where one edge in the BCTree (right) represents two edges (highlighted in blue) in the original Graph (left).

Thus, we propose six distinct orientation options for an edge between a cycle component and a cut vertex in the block-cut tree.

Definition 4.5 (Possible Orientations of an Edge in the Block-Cut Tree)**:** *Let $G = (V, E)$ be a cactus graph, and let BC denote its corresponding block-cut tree. Consider an edge $e = \{x, y\}$ in the block-cut tree, where x represents a cycle component in the original graph, and y is a cut vertex. To accurately capture the orientation of this edge in the block-cut tree, representing two edges in the original graph, we define the following possible orientations. If e is an edge connecting x to a cut vertex y on a higher layer, there are five feasible orientations.*

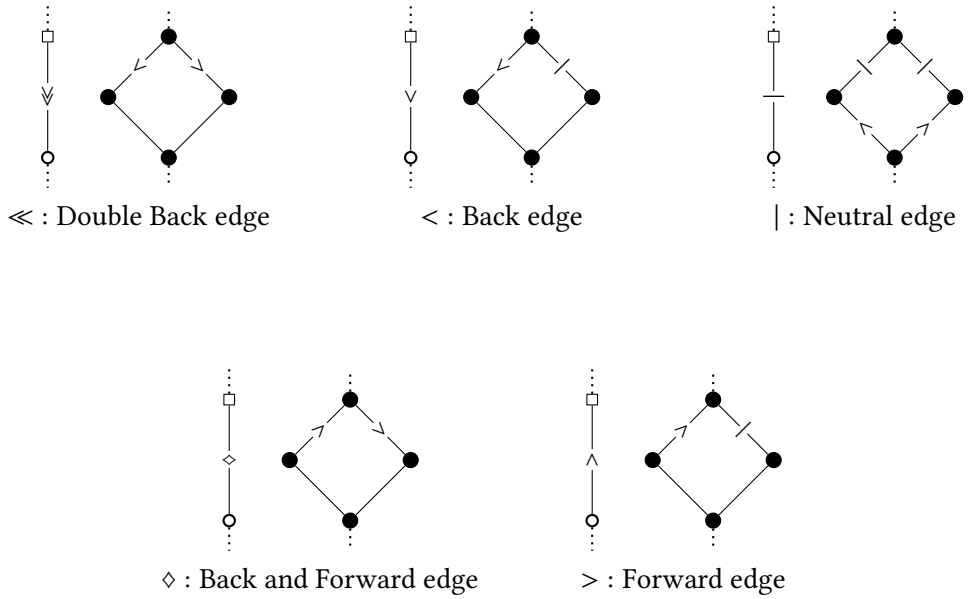


Figure 4.6: All five possible orientation of an edge in the block-cut tree when e connects a cycle component to a higher layer.

For the back and forward edges, it is irrelevant which of the two edges is the neutral edge and which is orientated (the same applies to the back and forward edge, where it is irrelevant which edge is forward and which is backward). The orientation double back edge is only feasible when the cycle is in the lower layer, while the orientation double forward edge is only applicable when the cycle is in the higher layer, as shown below. The other orientations $<, |, \diamond, >$ are feasable in either case.

Note that orienting an edge $e = \{x, y\}$ in one of the given directions, can be also described by the influence on the in, out degree of the vertices in the cycle of the original graph. In the following, we will use the notation $d^-(v)$ += 1 to indicate that the in-degree of vertex v in the original graph is incremented by 1, analogous to the operation used in programming. Let v be the vertex, whose both incident edges inside the cycle are effected by the orientation. Its neighbors in the cycle are referred to as u and u'.

- ***Double Back edge*** *(≪):* $d^+(v)$ += 2, $d^-(u)$ += 1, $d^-(u')$ += 1
- ***Back edge*** *(<):* $d^+(v)$ += 1, $d^-(u)$ += 1
- ***Neutral edge*** *(|): Does not affect either in or out degrees of either vertex.*
- ***Back and Forward edge*** *(◊):* $d^-(v)$ += 1, $d^+(v)$ y= 1, $d^-(u)$ += 1, $d^+(u')$ += 1
- ***Forward edge*** *(>):* $d^-(v)$ += 1, $d^+(u')$ += 1

If e is an edge connecting x to a cut vertex y on a lower layer, there are also five feasible orientations.

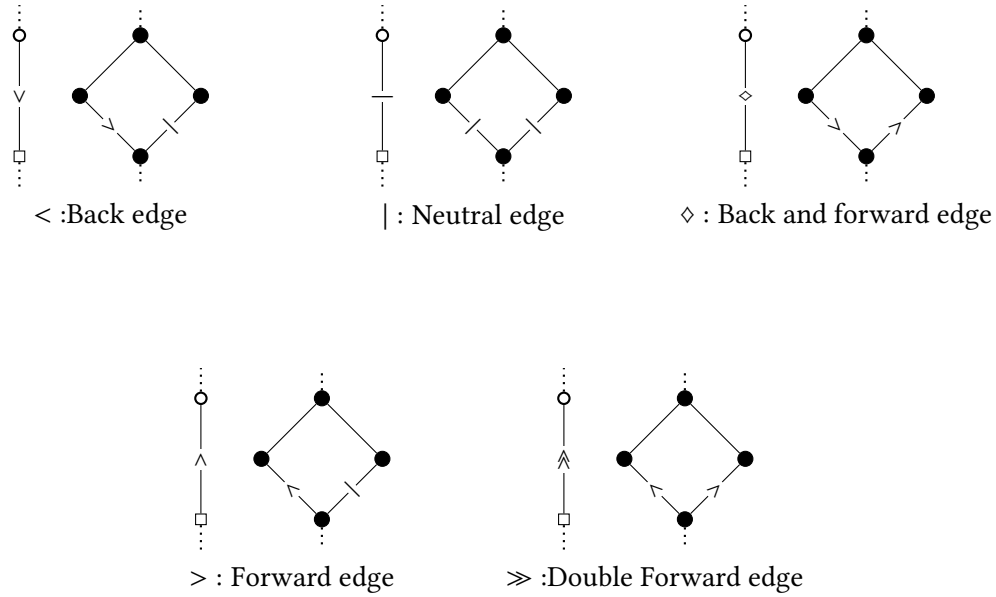


Figure 4.7: All five possible orientation of an edge in the block-cut tree when e connects a cycle component to a cut vertex on a lower layer.

The edge orientations can be described by the influence of the in- and out-degree of the vertices in the original graph as well. Let v be the vertex, whose both incident edges inside the cycle are effected by the orientation. Its neighbors in the cycle are referred to as u and u'.

- ***Back edge*** *(<):* $d^-(v)$ += 1, $d^+(u')$ += 1
- ***Neutral edge*** *(|): Does not affect either in or out degrees of either vertex.*
- ***Back and Forward edge*** *(◊):* $d^-(v)$ += 1, $d^+(v)$ += 1, $d^+(u)$ += 1, $d^-(u')$ += 1
- ***Forward edge*** *(>):* $d^+(v)$ += 1, $d^-(u)$ += 1
- ***Double Forward edge*** *(≪):* $d^+(v)$ += 2 $d^-(u')$ += 1, $d^-(u)$ += 1

By analyzing the effects on the in-degree and out-degree of the vertices, the infeasibility of double backward and double forward edges for each respective scenario can be explained. Specifically, if the cycle component is on a lower level, a double forward orientation of the corresponding edge would result in an increment of $d^-(y)$ by 2. This would contradict the requirements of a valid orientation.

These orientations allow the algorithm to accurately compute costs by taking the interactions between cycle blocks and cut vertices within the block-cut tree into account.

Note that within cycles, and when a cut vertex is connected solely to path blocks, only the basic orientations $<, |, >$ need to be considered. The effect on the in- and out-degree on the vertices in the original graph can be described analogously to the definition 4.5.

4.1.3 Cost Computation: Formal Introduction

We now turn to the computation of costs for each possible edge orientation in the graph. In the following, we will use the notations $c_{\ll}(e)$, $c_{<}(e)$, $c_{\diamond}(e)$, $c_{|}(e)$ $c_{>}(e)$ and $c_{\gg}(e)$ to represent the cost of orienting an edge e in the respective direction. Note that the costs are non-negative integers.

According to the definition of a valid orientation in the VOMO problem, a vertex either belongs to the origin (i.e., $d^-(v) = 0$) or must have an incoming edge from a neighboring vertex, implying $d^-(v) = 1$. Furthermore, once a vertex has an incoming edge, it can direct up to k outgoing edges, meaning $d^+(v) \leq k$. If a vertex is part of the origin, it is unrestricted in its outgoing edges, i.e., $d^+(v)$ can be arbitrary. Also there can not be a dependency cycle. However since we examine cactus graphs, with the property that two cycles can only share at most one vertex, this can easily be checked at the last edge of each cycle. Note that these restrictions have to be applied to the original graph.

As discussed previously, the algorithm examines the graph in a layered, bottom-up fashion using the block-cut tree. In each layer, the costs for the edges are computed based on valid combinations of ingoing and outgoing edges in respect of the definition of a valid orientation. To determine the validity of a given combination of edge orientations, we must translate the previously established restrictions to the BC-Tree, $BC = (V_{BC}, E_{BC})$. In this context, we denote the in-degree and out-degree of a vertex in the BC-Tree as d^-_{BC} and d^+_{BC}, respectively. Note that for any vertex $u \in V_{BC}$ representing a cut vertex $v \in V$, it holds that $d^+_{BC}(u) = d^+(v)$ and $d^-_{BC}(u) = d^-(v)$.

By ensuring that all edges between vertices in lower layers are processed prior to those in higher layers, the algorithm can efficiently compute costs for the orientations.

The valid combinations of in-degrees and out-degrees are formally defined, separated into two cases:

Definition 4.6 (Local Valid Orientation in a cut vertex u)**:** *Let $e = \{u, v\} \in E_{BC}$ be an edge in the BC-Tree, where u represents a cut vertex on the lower layer of the edge. A mapping $\pi_d : N(u) \to \{<, |, \diamond, >, \gg\}$ is called a local valid orientation for e being oriented $d \in \{<, |, \diamond, >, \gg\}$ in u, if the following conditions are satisfied:*

- $\pi_d(e) = d$
- *If v does not represent a cycle component, then $d \in \{<, |, >\}$,*
- *The orientation D of the subgraph $H = (N[u], \{(u, w), w \in N(u)\})$, where e is oriented as d and for all $w \in N(u) \setminus \{v\}$, $e' = \{w, u\}$ is oriented as $\pi_d(w)$, is subject to the following conditions:*
  - $d^+_{BC}(u) \leq 1$
  - $d^-_{BC}(u) = 1 \implies d^+_{BC}(u) \leq k$

We denote by $\Pi_d(e)$, for $d \in \{<, |, \diamond, >, \gg\}$, the set of all local valid orientations for $e = \{u, v\}$ being oriented according to d and u representing a cut-vertex component on the lower layer.

Definition 4.7 (Local Valid Orientation in a cycle C)**:** *Let $e = \{u, v\} \in E_{BC}$ be an edge of the BC-Tree, where u is the vertex on the lower layer and represents a cycle component. This cycle, denoted as C, with $n \in \mathbb{N}$ vertices is represented by the vertices $u_0, u_1, \ldots, u_{n-1}$ in the original graph, where u_0 is the cut vertex corresponding to v.*

A mapping $\pi'_d : \{u_0, u_1, \ldots, u_{n-1}\} \to \{<, |, >\}$ is called a local valid orientation for e, oriented according to $d \in \{\ll, <, |, \diamond, >\}$, if the following conditions are met:

- *$\pi'_d(u_0)$ and $\pi'_d(u_{n-1})$ specify the same orientation as defined in the corresponding cylce in figure 4.6 for orienting e according to d for the edge from the top vertex of the cycle to its neighbors.*
- *The orientation D of the cycle C, defined such that for all $i \in \{0, 1, \ldots, n-1\}$, $e' = \{u_i, u_{(i+1) \mod n}\}$ is oriented according to $\pi'_d(u_i)$ in that sense that a backward edge is orientated towards u_i and so on, is subject to the following conditions:*
  - *The orientation does not form a dependency cycle (see definition 4.1).*
  - *For all $i \in \{0, 1, \ldots, n-1\}$, $d^-(u_i) \leq 1$.*
  - *For all $i \in \{0, 1, \ldots, n-1\}$: $d^-(u_i) = 1 \implies d^+(u_i) \leq k$.*

We denote by $\Pi'_d(e)$, for $d \in \{\ll, <, |, \diamond, >\}$, the set of all local valid orientations where u is a cycle component and with e oriented according to d.

The reason no such definition is required for paths is that the BC-Tree already covers all edges from the original graph through the cases defined above. For further details, see Section 4.1.7.

A more in-depth understanding of these local valid orientations is provided in following examples below. The algorithm aims to identify the valid orientation with the lowest cost for each edge in the BC-Tree. The exact cost calculation depends on the type of each edge and is computed recursively.

4.1.4 Cost computation: Base case of the recursion

If the lower vertex of an edge e has no other incident edges, we can deduce that the lower vertex of e is a leaf of the BC-Tree.

Figure 4.8: The two options for leaf components in the block cut tree. The costs of the left edge are defined in defintion 4.8 whereas the costs of the right edge are defined in definition 4.9.

Since costs represent the number of vertices that must be part of the origin to produce a valid orientation in the subgraph below e and the subgraph below e only contains this leaf component, the costs for orientation e, for set k are always the same.

Definition 4.8: *Let $e = \{u, v\} \in E_{BC}$ be an edge of the BC-Tree with the vertex u in the lower layer representing a leaf of the original graph. We define the costs of orientating e as the following.*

- $c_<(e) = 0$
- $c_|(e) = 1$
- $c_>(e) = 1$

Definition 4.9: *Let $e = \{u, v\} \in E_{BC}$ be an edge of the BC-Tree with the vertex u in the lower layer representing a cycle with only one cut vertex of the original graph. If $k = 1$ We define the costs of orientating e as follows.*

- $c_\ll(e) = 0$
- $c_<(e) = 1$
- $c_|(e) = 1$
- $c_\diamond(e) = 1$
- $c_>(e) = 1$

Note that for the first case the costs apply to an arbitrary value of k, but for the second case the costs change. However, they can easily be computed as defined in Section 4.1.8.

4.1.5 Cost computation: e connecting a cut vertex to a path component

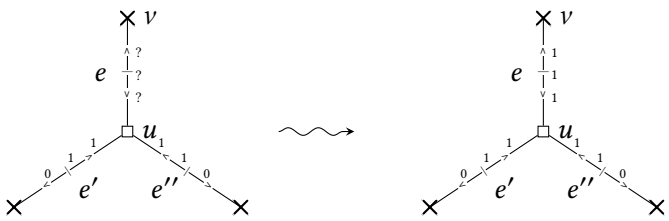


Figure 4.9: Example state before (on the left) and after (on the right) calculating the costs for different orientations of the edge e. The numbers next to the respective orientation represent the costs of orienting the edge in this direction.

To clarify the process of edge cost computation, consider the example shown in Figure 4.9. Recall that $c_<(e)$ represents the cost of orienting e as a backward edge, meaning the direction is from a higher layer to a lower one. Since u is on a lower layer than v, this corresponds to orienting the edge as $e = (v, u)$.

By setting e as a backward edge, u already has an in-degree of one, which, according to Definition 4.6, is the maximum allowable in-degree for any vertex. As a result, the remaining edges e' and e'' must either be neutral or backward edges. Given that $k = 1$, it follows from Definition 4.6 that $d^+(u) \leq 1$, meaning that only one of the edges can be a backward edge. Suppose we choose e' as a back edge and e'' as a neutral edge. The total cost for setting e as a backward edge is the sum of the costs for the chosen orientations of e' and e''. Specifically, $c_<(e) = c_<(e') + c_|(e'') = 0 + 1 = 1$.

For $c_|(e)$, we must ensure that u has one incoming edge, allowing it to have one outgoing edge. Suppose we choose e' to be a forward edge, directed towards u. In this case, e'' can serve as the outgoing edge from u. The corresponding cost calculation would be $c_|(e) = c_>(e') + c_<(e'') = 1 + 0 = 1$.

For $c_>(e)$, designating e as a forward edge, i.e., $e = (u, v)$, already accounts for one outgoing edge from u. Thus, the orientations of e' and e'' must involve a combination of one forward and one neutral edge, resulting in a cost of two. However, in this scenario, the algorithm would choose to include u in the origin, allowing all of u's edges to be treated as outgoing with an additional cost of one. Hence, $c_>(e) = c_<(e') + c_<(e'') + 1 = 0 + 0 + 1 = 1$. This illustrates that the cost of an edge reflects how many vertices below the edge must be part of the origin when it is oriented in a specific way.

Finally, recall that if two adjacent vertices are included in the origin, the edge between them would be oriented as a neutral edge, ensuring that $d^-(u) = 0$ holds for both vertices.

So, to compute the costs of a local valid orientation π_d for orientating e according to $d \in \{<, |, >\}$ whenever u is a cut vertex on the lower layer and v represents a path component on the higher layer we sum up the costs for orientating all incident edges of u (except for e) according to π_d and add one extra cost if $d^-_{BC}(u) = 0$, meaning u must be selected as part of the origin. Since the cost of orientating e according to d is the minimum cost of all $\pi_d \in \Pi_d(e)$ we can define:

Definition 4.10: *Let $e = \{u, v\} \in E_{BC}$ be an edge connection a cut vertex component u to a path component v in a higher layer. For $d \in \{<, |, >, \}$*

$$c_d(e) = \min_{\pi_d \in \Pi_d(e)} c(\pi_d)$$

with

$$c_d(\pi_d) = \begin{cases} \sum_{e' = \in E'} c_{\pi_d(e')}(e') & \text{if } d^-_{BC}(u) = 1 \text{ according to } \pi_d \\ \sum_{e' = \in E'} c_{\pi_d(e')}(e') + 1 & \text{if } d^-_{BC}(u) = 0 \text{ according to } \pi_d \end{cases}$$

and $E' = \{\{u, w\} \mid w \in N(u) \setminus \{v\}\}$.

4.1.6 Cost computation: e connecting a cut vertex to a cycle component

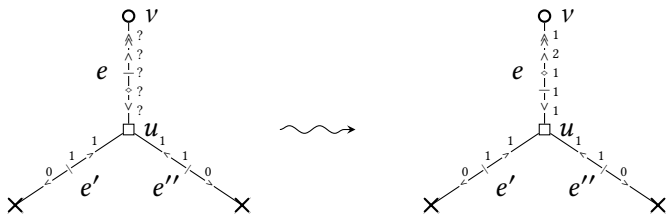


Figure 4.10: Example state before (on the left) and after (on the right) calculating the costs for different orientations of the edge e.

Now assume that v represents a cycle component instead of a path component, illustrated in Figure 4.10. We must consider that the edge e now corresponds to two edges, e_1 and e_2, in the original graph rather than just one.

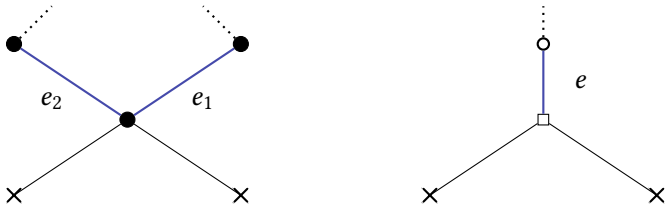


Figure 4.11: Example how two edged in the original graph are represented by just one edge in the BC-Tree

$c_<(e)$, $c_\diamond(e)$, $c_|(e)$ $c_>(e)$ and $c_\gg(e)$ are computed with the same considerations as before. Meaning we compute the costs of all possible local valid orientatios in u when $e1$ and $e2$ are directed according to the orientation of e and Defintion 4.5.

For example, orienting e as a backward edge translates to a neutral orientation of e_2, while orienting e_1 towards the cut vertex u. Note that in the context of the examination order, defined as clockwise within cycle components, this would be referred to as forward. To avoid misunderstandings, we describe the orientation in this explanation by specifying the vertex whose in , out degree is incremented in the original graph and the vertex towards which edge is oriented towards.

If e is oriented backwards, it could also translate to a neutral orientation of e_1 and an orientation of e_2 towards u. In both cases, the in-degree of u increases by one, and based on the previous analysis, either e' or e'' must be neutral, with the other edge either backward or neutral. By selecting the least costly valid orientation, we obtain $c_<(e) = c_<(e') + c_|(e'') = 0 + 1 = 1$.

$c_|(e)$, $c_>(e)$ can be calculated in the same way, resulting in the costs presented in Figure 4.10. In comparison to the calculations in Section 4.1.5, $c_\diamond(e)$ and $c_\gg(e)$ have to be computed, additionally.

First, we illustrate the calculation of $c_\gg(e)$: In the case where $k = 1$, edge e can only be double forward if u is part of the origin; otherwise, $d^+(v) > k = 1$. Thus, the total cost of this orientation must be incremented by 1. If u is part of the origin, e' and e'' be either neutral or back edges. Orienting both edges as back edges results in a minimal cost: $c_\gg(e) = c_<(e') + c_<(e'') + 1 = 0 + 0 + 1 = 1$.

Moreover, orienting e as a forward-backward edge restricts e' and e'' to neutral orientations, as any other orientation would violate the in-degree and out-degree requirements for a valid orientation. Therefore, the cost is: $c_\diamond(e) = c_|(e') + c_|(e'') = 1 + 1 = 2$

We can transfer the formal equation to calculate the costs in this case from 4.1.5:

Definition 4.11: *Let $e = \{u, v\} \in E_{BC}$ be an edge connection a cut vertex component u to a cycle component v in a higher layer. For $d \in \{<, \diamond, |, >, \gg\}$*

$$c_d(e) = \min_{\pi_d \in \Pi_d(e)} c(\pi_d)$$

with

$$c_d(\pi_d) = \begin{cases} \sum_{e' = \in E'} c_{\pi_d(e')}(e') & \text{if } d^-_{BC}(u) = 1 \text{ according to } \pi_d \\ \sum_{e' = \in E'} c_{\pi_d(e')}(e') + 1 & \text{if } d^-_{BC}(u) = 0 \text{ according to } \pi_d \end{cases}$$

and $E' = \{\{u, w\} \mid w \in N(u) \setminus \{v\}\}$.

After establishing how orientation costs are calculated when the cut vertex resides in the lower layer, we now examine cost calculations when the cut vertex is located on a higher layer. In this scenario, it is also essential to differentiate between cases where the lower component represents a cycle or a path.

4.1.7 Cost computation: e connecting a path component to a cut vertex.

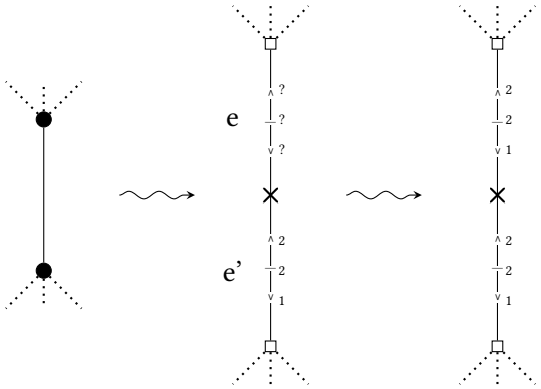


Figure 4.12: Example demonstrating how costs are mirrored from the edge of a cut vertex in a lower layer to a path block, to the edge from a path block to a cut vertex in a higher layer. This also illustrates that e and e' represent the same edge in the initial graph.

Note that each vertex representing a path component in the block-cut tree, that is not a leaf in the BC-Tree, has exactly two incident vertices. Both being cut vertices, one on a lower layer and the other on a higher layer. Since e and e' represent the same edge in the initial graph, the costs associated with both edges are identical.

Definition 4.12: *Let $e = \{u, v\} \in E_{BC}$ be an edge connection a path component u to a cut vertex v in a higher layer and $e' \in E_{BC}$ the unique edge connection u to a cutvertex v' in a lower layer . For $d \in \{<, |, >, \}$*

$$c_d(e) = c_d(e').$$

4.1.8 Cost computation: e connecting a cycle component to a cut vertex

When an edge connects a cycle component to a cut vertex in the block-cut tree, it is necessary to revert to the original representation of the cycle in order to accurately compute the costs associated with orienting this edge. This process is illustrated below, followed by an explanation of how the costs are computed in this scenario, providing an understanding how costs in cycles are computed in general. Note that for each edge only the feasible orientations are considered and thus computed.

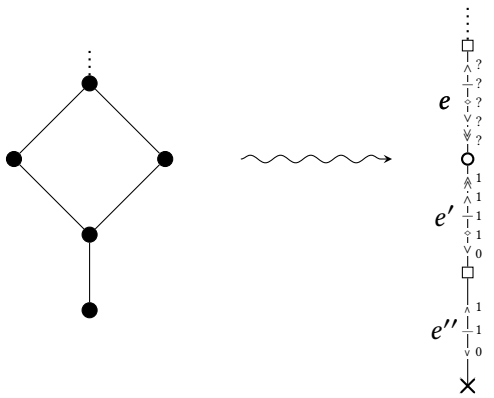
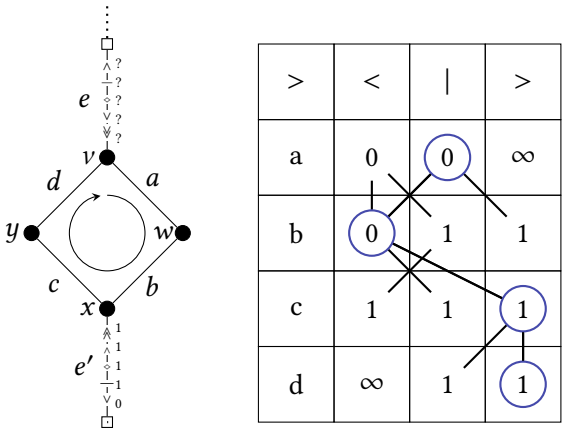


Figure 4.13: Example of a subgraph including a cycle component and the corresponding part of the block-cut tree.



>	<	|	>
a	0	0	∞
b	0	1	1
c	1	1	1
d	∞	1	1

Figure 4.14: Example of calculating the cost for orienting e as a forward edge, with $c_>(e) = 1$. Connections in the table demonstrate which preceding orientation is selected for each entry, with blue circles indicating the chosen orientations for each edge.

The method for calculating the costs of orienting the edge e'' and e' in 4.13 has already been introduced.

To better understand how the costs for orienting edge e are calculated when e connects a cycle component to a cut vertex on a higher layer in the block-cut tree (BC-Tree), refer to Figures 4.13 and 4.14. On the left side of Figure 4.14, we observe how the algorithm replaces the cycle vertex in the block-cut tree (from Figure 4.13) with the corresponding cycle from the original graph G (as shown on the left side of Figure 4.13).

The goal is to compute the costs for all five possible orientations of e. Figure 4.14 demonstrates the computation for $c_>(e)$, which represents the cost of orienting e as a forward edge. The algorithm traverses the cycle's edges in a clockwise direction, evaluating each edge's possible local valid orientations according to Definition 4.7 and ultimately selecting the cheapest option.

We begin with edge $a = \{v, w\}$. Since e is a forward edge, it follows from Definition 4.6 that edge a can either be neutral or backward. Since if the vertex v of the original graph, is part of the origin or not will be evaluated at the next step, when we compute the costs for the edge on the next layer incident to e, we start with costs $c_<(a) = 0$ and $c_|(a) = 0$.

Next, we consider edge $b = \{w, x\}$. Similar to the process in Figure 4.9, we evaluate the possible orientations of edge b to ensure that vertex w satisfies the conditions for a local valid orientation. By orienting b as a back edge, vertex w gains an in-degree of at least one, allowing it to have one outgoing edge but no additional incoming edges. In this case, edge a could either be neutral or backward, with the resulting cost being $c_<(b) = c_|(a) = 0$ (or equivalently $c_<(b) = c_<(a) = 0$). Orienting b as a neutral edge results in $c_|(b) = 1$, as w would have an in-degree of zero, requiring w to be added to the origin at an additional cost. Similarly, we compute $c_>(b)$.

For the subsequent edge $c = \{y, x\}$, we must consider the edge from the cut vertex on the lower layer e' in our computations. In the table from Figure 4.14, orienting edge c as a back edge is preceded by either a neutral or backward orientation for edge b, corresponding to the backward or forward-backward orientation of e'. The cost of this orientation of c is the minimum of all sums between the cost of orienting the predecessor edge inside the cycle (b) in a valid way (in this case either neutral or backward) and the cost of the corresponding orientation (in this case meaning backward or forward-backward) for the edge connecting to the lower layer (e').

Note that the orientation of the predecessor b is relative to the evaluation order in the cycle, whereas the orientation of e' is relative to the layered structure. Thus, we calculate: $c_<(c) = \min\{c_|(b) + c_<(e'), c_<(b) + c_\diamond(e')\} = \min\{1+0, 0+1\} = 1$

Since the costs are the same, both options are valid. For simplicity, the algorithm selects b as a neutral edge. For more information how the algorithm prioritizes certain predecessors over others 4.17., will later be discussed in the section about the algorithm time complexity.

A neutral orientation of c allows b to be oriented forward, neutral, or backward, and subsequently e' can be oriented backward, neutral, or forward. Choosing either a backward or forward orientation for e' results in a cost of 1, while a neutral orientation for b leads to a cost of 2. Both forward and backward orientations are valid, but the algorithm prefers a backward orientation to avoid dependency cycles. Therefore, we get $c_|(c) = c_<(b) + c_>(e') = 1$ and select a backward orientation for the predecessor b, which corresponds to a simple forward orientation for e'.

A forward orientation of c requires b to be oriented backward (double forward for e'), forward (back and forward for e'), or neutral (forward for e'). Since all of combination 2, but orienting b as a back and thus e' as a double forwad edge, we choose this combination and get: $c_>(c) = c_<(b) + c_\gg(e') = 1$.

For the last edge d, we apply the same method but with additional considerations, as this is the closing edge of the cycle. Specifically, we ensure that the chosen orientations do not create a dependency cycle and that only the orientations allowed by Definition 4.6 are selected. Recall that since we compute the cost for e to be a forward edge, d can only be neutral or forward.

Finally, the cost $c_>(e)$ is determined by the minimum value in the last row, which is 1. Since two equal values exist, we have a choice between them. By backtracking, we determine the orientations of the other edges in the cycle. In this case, we ensure that edges a and d are oriented such that vertex v has one incoming edge and one neutral edge. The algorithm applies this process for all five orientations of e, storing the corresponding costs and edge orientations within the cycle for each case.

To formalize this complex procedure, recall the definition of a local valid orientation in a cycle (Definition 4.7). Similar to the computations in previous cases, we examine all local valid orientations for orientating e according to $d \in \{\ll, <, |, \diamond, >\}$ and choose the orientation $\pi'_d \in \Pi'_d$ with the lowest cost. Those costs are influenced by two factors. First, the number of vertices inside the circle which must be part of the origin in each respective case.

Secondly, the incident edges to the cycle component in the BC-Tree: We sum up all the costs for the orientations implied by π'_d according to Definition 4.5.

Note, that each edge incident to the cycle component in the BC-Tree corresponds to a cut vertex.

Definition 4.13: *Let $e =$ be an edge in the BC-Tree connecting a cycle component u, representing a cycle C with length $n \in \mathbb{N}$, to a cut vertex v in a higher layer. The vertices of C are called $\{u_0, u_1, \ldots, u_{n-1}\}$ and v represents the cut vertex u_0. We define the mapping $\gamma : \Pi'_d \times N(u) \to \{<, |, \diamond, >, \gg\}$ that assigns each neighbor w from u in the BC-Tree the orientation of the edge $\{u, w\}$, according to the orientations specified by the local valid orientation for the cycle represented by u. We set $o(\pi'_d) \in \mathbb{N}$ as the number of vertices in $C \setminus \{u_i \in V_C \mid u_i$ is a cut vertex.$\}$ that have no edges oriented towards them in the orientation π'_d and are not a cut vertex in C. For $d \in \{\ll, <, |, \diamond, >\}$*

$$c_d(e) = \min_{\pi'_d \in \Pi'_d(e)} c(\pi'_d)$$

with

$$c(\pi'_d) = o(\pi'_d) + \sum_{w \in N(u) \setminus \{v\}} c_{\gamma(\pi'_d, \{u,w\})}(\{u, w\})$$

4.1.9 Processing the last edge

Finally, when the algorithm processes the last edge in the block-cut tree (BC-Tree)—the edge connecting a cut vertex to the path component containing the earlier added root r—it computes the cost of setting this edge as neutral and selects the final orientation for all preceding edges. The algorithm then backtracks to apply the chosen orientations to all edges of the original graph. Once this step is complete, the root r is removed, leaving only the vertices V of G.

For illustration, consider the figure below, which shows the final computation of the last edge and the selected orientations for the original graph G from the example 4.4.

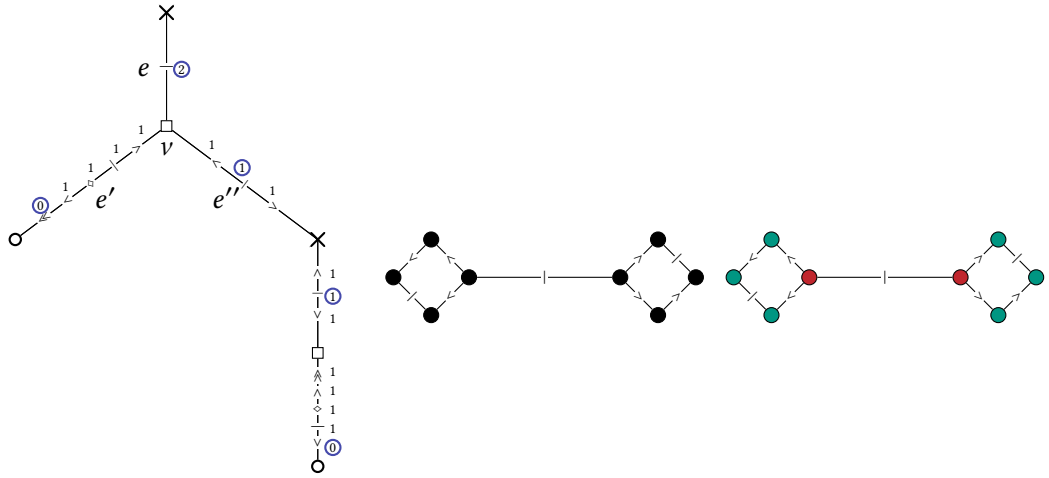


Figure 4.15: On the left, the final costs for each edge orientation in the BC-Tree are shown, with the selected orientations highlighted in blue. The middle image represents the corresponding graph with the resulting valid orientation. On the right, the implied solution for the k-Power Dominating Set (k-PDS) is displayed, with the origin O, marked in red, as the k-Power Dominating Set.

In this example, the cost for setting edge e as neutral is $c_|(e) = 2$. Any configuration other than choosing e' as a double back edge results in a cost of two. Given that $k = 1$, only vertices in the origin O are allowed to have an out-degree greater than one. Thus, the cut vertex v must be added to the origin O, and the cost is again two when e' is selected as a double back edge. Specifically, $c_|(e) = c_{\ll}(e') + 1 + c_|(e'') = 0 + 1 + 1 = 2$.

Additionally, the graphic illustrates the selected orientations in the BC-Tree, as well as the resulting valid orientation of the original graph. Through iterating over the edges, the algorithm identifies each vertex with an in-degree of zero and adds it to the origin O.

4.1.10 Pseudocode

Algorithm 4.1: k-PDS Cactus

Input: Cactus Graph G
Output: Minimum Power Dominating Set O

```
Initialize an empty Set S
Add root r with edge {r, v} to an arbitrary vertex v ∈ V to the graph
Transform Cactus Graph into corresponding Block-Cut Tree (BC-Tree)
Initialize a queue of the edges of the BC-Tree (rooted at r) in a reverse topological order
for each edge e in the queue do
    if e connects to the path component including the root r then
        calculate c_|(e) and set orientation of edges in initial Graph G
        break
    if e connects a cycle component to a cut vertex on a higher layer then
        Calculate c_≪(e), c_<(e), c_|(e), c_◇(e), c_>(e)
    else if e connects a cut vertex to a cycle component on a higher layer then
        Calculate c_<(e), c_|(e), c_◇(e), c_>(e), c_≫(e)
    else
        Calculate c_<(e), c_|(e), c_>(e)
for each vertex v in the initial Graph G do
    if d⁻(v) == 0 then
        Add v to O
return O
```

4.2 Proof of correctness

In this part we prove the correctness of the algorithm.

Theorem 4.14: *The k-PDS Cactus Algorithm solves the k-Power Dominating Set problem on Cactus Graphs.*

To establish correctness, we will show that at each step of the dynamic program, the current solution adheres to a valid orientation in the original graph. Furthermore, we will demonstrate that the solution is optimal, meaning that at each step, the number of vertices included in the origin—of vertices for which all incident edges have been processed—is minimal. This ensures that the algorithm successfully solves the Valid Orientation with Minimum Origin (VOMO) problem on arbitrary cactus Graph instances. Together with Theorem 4.3, this proves Theorem 4.14.

Proof. Let $G = (V, E)$ be a cactus graph, and let BC be its corresponding block-cut (BC) tree.

We begin by considering the case where a block is a leaf node in the BC-Tree.

Case 1.1: The leaf is a path block. Let u, v be the vertices of the path block p in the BC-Tree, which represents the edge $e = \{u, v\}$ in the original graph G. Since p is a leaf node in the BC-Tree, one of the two vertices must be a leaf in the original graph, while the other must be a cut vertex. Without loss of generality, let u be the leaf vertex and v be the cut vertex.

According to the definition of a valid orientation (Definition 4.2), (for any $k \in \mathbb{N}_0$) the vertex u must either have an indegree $d^-(u) = 1$, or an indegree $d^-(u) = 0$, in which case it must be part of the origin.

We now analyze the different possible orientations of the edge e in relation to the vertices u and v:

- If we orient e as (u, v), meaning the edge is directed towards v, this corresponds to a back edge in the BC-Tree. In this case, v will not be part of the origin, as it receives the incoming edge.

- If e is undirected (neutral) or oriented as (v, u) (forward edge in the BC-Tree), then u will have an in degree of zero. Consequently, u must be part of the origin.

In each scenario, we have considered all possible valid orientations of the edge e with respect to vertex u, ensuring that the orientation satisfies the conditions outlined in the definition of a valid orientation.

We conclude that the costs defined in Definition 4.8 correctly represent the number of vertices that must be included in the origin based on the chosen orientation of the edge e.

Case 1.2: The leaf is a cycle block. Let C be a cycle component with n vertices in the original graph G. Assume the cycle is embedded with vertex v at the top, such that v is the only vertex in C with an incident edge e connecting it to a cut vertex on a higher layer in the BC-Tree. Let the two incident edges within the cycle be denoted by e_1 and e_n.

For each of the edges e_1 and e_n, the possible orientations include directing them towards v, away from v, or leaving them undirected. This gives nine possible combinations for the orientations of e_1 and e_n. However, orienting both edges towards v would violate the condition $d^-(v) \leq 1$, since $d^-(v) = 2$ is not allowed. Thus, we are left with eight valid combinations of orientations for e_1 and e_n, as described in Definition 4.5.

The algorithm computes all eight valid combinations. For each of these, the algorithm further considers the three possible orientations for each edge within the cycle (forward, backward, or neutral). This ensures that every possible configuration of the cycle is evaluated. For each edge $\{u, v\}$ within the cycle, the algorithm follows a clockwise set embedding and checks all orientations, respecting the degree constraints at each vertex.

Since the cost at each step is computed taking the previous solution and adding one, the current vertex (first vertex of the current edge) is added to the origin, the correctness of the cost calculation follows by induction. At the final edge of the cycle, the algorithm checks for dependency cycles and discards any configurations that form such cycles.

In summary, the algorithm explores all valid orientations for the cycle, selecting the configuration with the minimal cost. Note that the algorithm does not need to store all orientations, a topic that will be discussed further in the time complexity analysis in the next section.

Additionally, note that there are always at least two possible orientations for the edge e with finite costs—namely, the double-back and neutral orientations. In these cases, the cycle cannot form a dependency cycle. By orienting all other edges in the cycle as neutral, we obtain a valid orientation.

Now, we turn to the case where multiple edges from different components connect to a single cut vertex on a higher level. We will show that the algorithm correctly calculates the costs for each possible orientation of the edge $e = \{u, v\}$, where u is the cut vertex in the BC-Tree and v is a component on a higher layer.

Case 2: $e = \{u, v\}$ is an edge from a cut vertex u to a block component v on a higher layer in the BC-Tree. Here, we assume that the costs of each valid orientation for the edges from components on lower layers have already been calculated correctly. For each possible orientation of the edge (forward, neutral, or back in the case of a path block, and additionally back-and-forward or double-forward for a cycle block), the algorithm selects the least costly solution from all valid orientations. Correctly accounting for the influence of orienting e in each respective direction on the degree of u, the algorithm calculates the sum of all chosen orientations. If u has an indegree of zero, an additional cost of one is included. Thus this step is correct. Furthermore, the algorithm does not need to compute all possible orientations explicitly at this stage. Additional details on this optimization are discussed in the following section.

Next, we examine the case where the algorithm computes the costs of an edge from a block component to a cut vertex on a higher layer in the block-cut tree, allowing for edges between cut vertices from lower layers and the block component.

Case 3: $e = \{u, v\}$ is an edge from a block component u to a cut vertex v on a higher layer in the BC-Tree.

First, consider the scenario in which u is a path block. As illustrated in Figure 4.12, both the edge $e' = \{x, u\}$, connecting a cut vertex on a lower layer to the path block, and $e = \{u, v\}$, represent the same edge in the original graph—namely, the edge connecting the two vertices within the path block. Since the costs associated with this edge have already been computed correctly for e', the algorithm ensures correctness by mirroring these costs for e. If u is a cycle, the argument follows similarly to Case 1.2. All possible orientations of edges within the cycle are considered, ultimately selecting the orientation with the lowest cost for each possible orientation of e. Additionally, the cost of an edge $e' = \{x, u\}$, where x is a cut vertex in a lower layer of the BC-Tree, already accounts for whether the corresponding vertex within the cycle in the original graph is part of the origin. Thus, the algorithm correctly computes the costs by adding the orientation cost of e' with that of the preceding edge's orientation (evaluated in a clockwise order in a set embedding) according to the combinations outlined in Definition 4.7. This approach ensures that all valid orientations are evaluated. Consequently, this step of the algorithm is also correct.

Finally, we show that the neutral cost of the edge connected to the path with the root represents the minimum number of vertices in the origin and that the resulting orientation is valid in the graph G. Observe that the added root r, with its single edge $e = \{r, v\}$ where $v \in V$, is always a leaf in the BC-Tree, as it has only one incident edge by definition. Consequently, there is only one edge in the highest layer of the BC-Tree, which connects the cut vertex to the path with the root. Since we have demonstrated that the algorithm accurately calculates the costs at each step, it also computes the correct cost for this final edge.

Moreover, r is neither part of a cycle nor connected to any component or cut vertex in a higher layer, meaning r will not be included in the origin. By orienting this edge as a neutral edge, the degree of v remains unaffected. Thus, the resulting orientation in the original graph G is a valid orientation with the minimum origin. Together with Theorem 4.3, this proves the claim.

∎

4.3 Time Complexity

In the following, we analyze the time complexity of the k-PDS-Cactus algorithm. Guo et al. [GNR08] established that the Power Dominating Set (PDS) problem, and by extension, the k-PDS problem, can be solved in $O(c^{t^2 \cdot n})$ time, where t represents the treewidth of the graph and n the number of vertices. Since cactus graphs have a treewidth of 2, our objective was to develop a linear-time algorithm specifically tailored to cactus graphs. This focus leads us to the following theorem.

Theorem 4.15: *The k-PDS-Cactus Algorithm has a time complexity of $O(n)$.*

To establish this complexity result, we first introduce two supporting lemmas that will later be used in proving Theorem 4.15.

Lemma 4.16: *Orienting an edge e as a forward edge is never cheaper than orienting it as a neutral edge. Formally, $c_|(e) \leq c_>(e)$.*

Proof. Suppose, for contradiction, that there exists an edge $e = \{u, v\}$ for which the claim does not hold, with e being the first such edge in the algorithm. Observe that this cannot occur in the base case, where e is incident to a leaf vertex in the BC-Tree. Also, when u is part of the origin, the cost for both orientations is identical by definition, contradicting the assumption.

In the case where u is not in the origin, it must have at least one incoming edge, say e'. If e is oriented as a forward edge, only $k-1$ of the remaining edges incident to u from lower layers, apart from e', can be oriented as back edges; the rest must be neutral. However, if e is oriented as a neutral edge, it is feasible to orient the other incident edges from lower layers as either neutral or back edges, allowing up to k edges to be back edges. Since the set of available orientations in the neutral case is a superset of the forward case, this contradicts the assumption. Thus, the lemma holds. ∎

For the next part, we introduce the method by which the algorithm greedily selects the orientations of predecessor edges in a cycle.

Definition 4.17 (Greedy Selection)**:** *When determining the cost $c_d(e)$ for an orientation $o \in \{<, |, >\}$ of an edge e in a cycle, if multiple predecessor orientations yield the same minimal cost, the algorithm prioritizes predecessors according to the following order. Let e be an edge with e' as its predecessor in the cycle.*

- *If e is a forward edge, we prefer e' to be a back edge over a neutral edge, and a neutral edge over a forward edge.*
- *If e is a neutral edge, we prefer e' to be a neutral edge over a back edge, and a back edge over a forward edge.*
- *If e is a back edge, we prefer e' to be a neutral edge over a back edge.*

Note that in the last case, e' cannot be a forward edge, as this would violate the definition of a valid orientation.

Lemma 4.18: *If there exists a valid combination of edge orientations with minimal cost in a cycle with the first and last edges of the cycle fixed, the greedy selection will identify it.*

Proof. Assume that the greedy selection does not find a cheapest combination of edge orientations in a cycle, even though one exists. Let e be the first edge in the cycle that prevents the cycle from forming a dependency cycle; that is, e is either oriented in the opposite direction from its predecessor or is the second consecutive neutral edge. Let e' denote its predecessor.

By definition, the greedy selection will only choose this orientation of e' if it would lead to a lower cost. It is straightforward to observe that adding the last vertex of the cycle to the origin prevents the cycle from forming a dependency cycle (except in the special case of a three-vertex cycle in a back-and-forward orientation, which has no valid orientation and is therefore irrelevant).

Consequently, the greedy selection adds the last vertex to the origin, which increases the cost by only one (even if there are incident edges from cut vertices on a lower level, which directly follows from Lemma 4.16). Since this cost was accounted for earlier, the greedy selection has indeed identified a cheapest combination of edge orientations that avoids a dependency cycle, leading to a contradiction. ∎

With these preparations, we now proceed with the proof of Theorem 4.15.

Proof. Adding a vertex and an edge to the graph has a constant time complexity. Note that for each of the following observations we use the fact that the number of edges in cactus graphs is bounded by $\lfloor 3n/2 \rfloor$, so $O(m) = O(n)$. Using Tarjan's Algorithm [Tar72] to find articulation points (or cut vertices) through a depth-first search (DFS), we can construct a block-cut (BC) tree in $O(n)$; see [Wes01] for further details.

Next, we examine the cost computation within the algorithm. Since the algorithm evaluates at most eight orientations per edge, it suffices to show that the cost of each orientation is bounded by a constant.

In cases where we compute orientations for an edge of a cycle with $l \in \mathbb{N}$ vertices connected to a single higher-layer cut vertex, we must construct five distinct $l \times 3$ tables, as illustrated in Figure 4.14. Checking for dependency cycles across all possible combinations of orientations in the cycle could theoretically have exponential complexity, as there can be up to$O(3^l)$ possible orientation combinations. However, by using the greedy selection strategy and Lemma 4.18, we reduce this check to only three combinations. Given that checking for dependency cycles is $O(l)$, this part remains within the linear time bound.

Computing orientations for an edge where a path block is connected to a cut vertex has constant time complexity.

Finally, consider the scenario where a cut vertex u, with multiple incident edges from lower layers, connects to a higher-layer component. Here, we use the following selection approach. If an indegree is required, we examine all incident edges from lower-layer components, orienting the first edge found with a forward orientation if this orientation is its cheapest option. By Lemma 4.16, no other edge can result in a cheaper final cost if set as a forward edge. If no minimal-cost forward edge is found, and $k > 0$ we search for a minimum-cost back-and-forward edge in the same way. This is done after evaluating cheap forward edges, since each back-and-forward edge increases u's outdegree by one, it allows one fewer back edge connection to lower-layer components.

Orienting an edge as a back or a double back edge is, at most, one cost cheaper than setting it as a neutral edge, this follows by the same argumentation of Lemma 4.16, as adding the current vertex to the origin for one cost contradicts the assumption of the opposite. To maintain the minimal cost, we iterate through the other edges, incident to u, selecting back edges where they yield a lower cost than neutral orientations, constrained by u's possible outdegree relative to the parameter k. For any remaining edges, if the outdegree limit, has not been reached, double-back orientations are chosen if they are cheaper than neutral orientations, with all other edges set as neutral.

This method ensures minimal cost, and each edge computation remains bounded by a constant, since in the worst case, we have to loop through all each edge, five times.

In the final step of the algorithm, each predecessor is assigned its orientation by selecting the final orientation. This allows the algorithm to backtrack through each orientation and check for vertices with an indegree of zero. Since this involves iterating once over all edges, this step is also $O(n)$.

Consequently, the k-PDS-Cactus algorithm achieves linear time complexity. ∎

5 Experiments

5.1 Experiment Setup

The experiments were conducted on a machine running Ubuntu 24.04.1 LTS with Linux kernel version 6.8. This system was equipped with an AMD Ryzen 5 5600X 6-Core Processor, providing 6 cores with 2 threads per core (totaling 12 threads) at a clock speed of 3.7 GHz, and 16 GB of RAM. Each test was executed 10 times, with the median time recorded to ensure consistent results.

5.2 Evaluation of the k-PDS-Cactus Algorithm

We begin by evaluating the k-PDS-Cactus Algorithm, as introduced in Section 4, on a range of cactus graphs sourced from the Pandapower dataset [Thu+18]. For comparison purposes, we used the MILP Solver developed by Bläsius and Göttlicher [BG23]. For instances of k-Power Dominating Set, with $k = 1$ (PDS), the MILP Solver was tested both with and without the application of our reduction rules. For values of $k > 1$, we applied our reduction rules, noting whether the instances were fully solved or specifying the number of remaining unobserved vertices.

	$k=1$						$k=3$					$k=5$				
Instance	S		**Red+**S		**Algo**		**Red**			**Algo**		**Red**			**Algo**	
	γ_P	t	γ_P	t	γ_P	t	S	UO	t	γ_P	t	S	UO	t	γ_P	t
Cactus10	2	6	2	2*	2	50μ	1	0	0.1	1	51μ	1	0	0.1	1	53μ
Cactus100	16	53	16	2*	16	0.5	7	0	0.1	7	0.5	2	0	0.1	2	0.5
Cactus500	79	1087	79	3	79	2.7	31	0	0.1	31	2.9	12	0	0.1	12	2.8
Cactus1000	147	9968	147	4	147	5.1	58	0	0.4	58	5.2	24	0	0.2	24	5.4
Cactus10000	1565	0.18h	1565	21	1565	61	561	5	8	561	62	230	0	5	230	64
Cactus20000	3097	3h	3097	38	3097	119	1129	10	15	1129	120	497	0	12	497	125
Cactus50000	-	>24h	7785	109	7785	322	2877	15	44	2875	322	1222	0	36	1222	321
Cactus100000	-	>24h	15430	204	15430	650	5674	13	97	5678	649	2477	12	80	2481	651

Table 5.1: Evaluation of the K-PDS-Cactus Algorithm for $k = 1$, $k = 3$, and $k = 5$, comparing runtime for $k = 1$ with the MILP Solver of Bläsius and Göttlicher [BG23], both with (Red+S) and without (S) the application of reduction rules. Time t is reported in milliseconds, microseconds (μ), or hours (h). γ_P denotes the size of the k-Power Dominating Set, S represents the number of pre-selected vertices, and UO indicates the count of unobserved vertices. The solver did not reach an optimal solution for the Cactus50000 and Cactus100000 instances within 24 hours.

The results confirm that the algorithm exhibits linear time complexity, though as noted by Guo et al. [GNR08], the constant factor hidden within the Big O notation remains substantial. Additionally, the reductions are highly effective on cactus graphs, simplifying them to the point where even brute-force solutions become feasible. Although achieving a linear-time solution for the K-PDS problem on cactus graphs is theoretically valuable, the introduced reduction rules demonstrate an even greater practical utility.

5.3 Evaluation of the Reduction Rules

As observed in previous experiments, the reduction rules are highly efficient. We further tested these reductions on a broader selection of instances from the Pandapower dataset [Thu+18], as well as on the Eastern, Western, Texas, and US instances from the Pandapower SimData set [Xu+20], representing key electrical networks in the US.

Our evaluation metrics include the number of vertices (n) and edges (m) remaining after applying the reductions, the count of pre-selected vertices (X), the count of excluded vertices (Y), and the number of unobserved vertices (UO), across various k values. Additionally, we assessed the impact of the Merge Low Degree (MLD) reduction on instances not fully solved, by testing the rules on $k = 2$ without it, providing insight into the effect of this newly introduced rule.

The reduction rules were applied in the following order: local rules were executed exhaustively to minimize computational costs, followed by global rules in this sequence: MLD, CL, Isol, ObsNP, ObsE, Dom, NecN. Minor changes in runtime were observed when adjusting the order of application.

	Input		$k = 1$						$k = 2$					
Instance	n	m	n	m	X	Y	UO	t	n	m	X	Y	UO	t
Illinois200	200	245	20	0	20	0	0	0	8	0	8	0	0	0
6515rte	6515	8104	908	267	742	130	130	8	449	234	311	117	102	6
9241pegase	9241	14207	2685	3474	680	1400	1700	51	756	1732	260	473	440	25
Texas	2000	2667	1144	1600	103	617	880	12	62	0	62	0	0	1
Western	10024	12241	4755	5946	658	2189	3434	107	231	0	231	0	0	8
Eastern	70047	83366	38326	48815	3167	16178	31735	4764	1317	0	1317	0	0	69
USA	82071	98291	44211	56343	3929	18982	36028	6077	1610	0	1610	0	0	79

	Input		$k = 5$						$k = 2$ w/o MLD					
Instance	n	m	n	m	X	Y	UO	t	n	m	X	Y	UO	t
Illinois200	200	245	2	0	2	0	0	0	-	-	-	-	-	-
6515rte	6515	8104	48	0	48	0	0	4	485	272	311	153	138	13
9241pegase	9241	14207	25	0	25	0	0	8	836	1814	260	553	520	39
Texas	2000	2667	24	0	24	0	0	0	-	-	-	-	-	-
Western	10024	12241	48	0	48	0	0	3	-	-	-	-	-	-
Eastern	70047	83366	174	0	174	0	0	34	-	-	-	-	-	-
USA	82071	98291	246	0	246	0	0	40	-	-	-	-	-	-

Table 5.2: Evaluation of reduction rules for various instances from the Pandapower set and specific US-based instances. The table provides results for $k = 1, 2, 5$, and for $k = 2$ without the Merge Low Degree (MLD) reduction rule to assess its impact. Metrics include the reduced number of vertices (n) and edges (m), pre-selected vertices (X), forbidden vertices (Y), unobserved vertices (UO), and processing time (t) in milliseconds. Dashes represent no data.

As anticipated, the reduction rules scale effectively with increasing values of k, efficiently solving most instances for $k = 2$ in under 100 milliseconds, and substantially reducing the remaining instances. An additional noteworthy observation is that applying the reduction rules without the MLD rule reduces the instances to a similar number of undecided vertices ($n - X - Y$), while being more time efficient. This arises because the MLD reduction shares similarities with the global domination rule, identifying non-essential vertices in an optimal solution but doing so with greater effectiveness.

6 Conclusion

In this thesis, we explored the k-Power Dominating Set (k-PDS) problem, extending beyond the traditional Power Dominating Set (PDS) problem by integrating a variable parameter, k, that modifies the propagation rule. This investigation covered both theoretical advancements in reduction rules applicable to k-PDS, along with the development and analysis of an efficient algorithm tailored to solve k-PDS on cactus graphs with linear time complexity.

In the first part our primary focus was on formulating reduction rules for k-PDS, adapted from or extending those used in classical PDS contexts, which proved remarkably efficient across a variety of test instances. These reduction rules demonstrated strong potential for pre-processing, reducing problem size significantly and enabling manageable solution spaces for instances that would otherwise be computationally challenging. On most cases they even outperformed the algorithm tailored for cactus graph, fully solving most instances, suggesting that the reduction-first approach can be superior in practice.

Additionally, while algorithms for k-PDS may need to be specifically tailored to graph classes with defined tree widths to guarantee exact solutions, the presented reduction rules provide an efficient and versatile alternative. These reductions broaden the potential for practical application across a range of graph structures.

6.1 Future Work

Moreover, the KPDS-Cactus Algorithm introduced in this thesis presents an opportunity to derive new reduction rules by addressing cactus graph attached within larger instances. Exploring whether this approach effectively simplifies larger graph configurations could be a interesting task in the future.

This approach also suggests a pathway for developing additional k-PDS algorithms specific to other graph classes, not only as solutions but as foundational elements for further reduction rules.

In conclusion, this work offers significant contributions to both the theoretical and practical understanding of k-PDS. The proposed reduction rules enable faster computations while revealing useful structural simplifications that may inspire new algorithmic developments. Future research could extend these rules to additional graph types, enhancing their applicability and impact on broader network configurations.

Bibliography

[BG23] Thomas Bläsius and Max Göttlicher. *An Efficient Algorithm for Power Dominating Set*. 2023. arXiv: *2306.09870*.

[CDMR12] Gerard Jennhwa Chang, Paul Dorbec, Mickael Montassier, and André Raspaud. "Generalized power domination of graphs". In: *Discrete Applied Mathematics* Volume 160 (2012), pp. 1691–1698. DOI: *https://doi.org/10.1016/j.dam.2012.03.007*.

[Die17] Reinhard Diestel. *Graph Theory*. 5th. Springer-Verlag, 2017. ISBN: 978-3-662-53621-6.

[GNR08] Jiong Guo, Rolf Niedermeier, and Daniel Raible. "Improved Algorithms and Complexity Results for Power Domination in Graphs". In: *Algorithmica* Volume 52 (2008), pp. 177–202.

[HLPT07] Wing-Kai Hon, Chih-Shan Liu, Sheng-Lung Peng, and Chuan Tang. "Power Domination on Block-cactus Graphs". In: (Jan. 2007).

[Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* Volume 1 (1972), pp. 146–160.

[Thu+18] Leon Thurner, Alexander Scheidler, Florian Schäfer, Jan-Hendrik Menke, Julian Dollichon, Friederike Meier, Steffen Meinecke, and Martin Braun. "Pandapower—An Open-Source Python Tool for Convenient Modeling, Analysis, and Optimization of Electric Power Systems". In: *IEEE Transactions on Power Systems* Volume 33 (2018), pp. 6510–6521. DOI: *10.1109/TPWRS.2018.2829021*.

[Wes01] Douglas B. West. *Introduction to Graph Theory*. 2nd. Upper Saddle River, NJ: Prentice Hall, 2001. ISBN: 978-0-13-014400-3.

[Xu+20] Yixing Xu, Nathan Myhrvold, Dhileep Sivam, Kaspar Mueller, Daniel J. Olsen, Bainan Xia, Daniel Livengood, Victoria Hunt, Benjamin Rouillé d'Orfeuil, Daniel Muldrew, Merrielle Ondreicka, and Megan Bettilyon. "U.S. Test System with High Spatial and Temporal Resolution for Renewable Integration Studies". In: *2020 IEEE Power Energy Society General Meeting (PESGM)*. 2020, pp. 1–5. DOI: *10.1109/PESGM41954.2020.9281850*.