Solving Power Dominating Set with Branch and Bound

Master's Thesis of

Julia Lutzweiler

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

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	PD Dr. Torsten Ueckerdt
Advisors:	Max Göttlicher, M.Sc.

01.12.2022 – 01.06.2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie.*
Karlsruhe, 01.06.2023

. . . .
(Julia Lutzweiler)

Abstract

Power dominating set is a graph problem that asks for the smallest subset of vertices that can observe all vertices. Observation is defined by two rules. The first one states that all neighbors of the power dominating set are observed. This is identical to dominating set, of which power dominating set is an extension. The second rule allows for propagation of the observation status to vertices that are the only unobserved neighbor of an observed vertex. Like dominating set, power dominating set is NP-complete.

Multiple approaches for solving power dominating set have been presented across literature, including a formulation as an integer linear program. In this thesis, we present an algorithm based on branch and bound. We derive a variety of lower and upper bounds to the optimal solution and relate power dominating set to a graph covering problem. Moreover, we discuss different strategies for branching. The implementation of our algorithm outperforms the reference solution based on an integer linear program for small graphs but is slower on larger instances.

Zusammenfassung

Power Dominating Set ist ein Graphenproblem, bei dem die kleinste Teilmenge der Knoten zu finden ist, sodass alle Knoten beobachtet werden. Die Beobachtung von Knoten ist durch zwei Regeln definiert. Erstere besagt, dass die Nachbarn des Power Dominating Set beobachtet sind. Diese Regel ist identisch zu Dominating Set, ein Problem das Power Dominating Set erweitert. Die zweite Regel ermöglicht es, weitere Knoten zu beobachten: Jeder Knoten, der der einzige unbeobachtete Nachbar eines beobachteten Knotens ist, wird auch beobachtet. Power Dominating Set ist wie auch Dominating Set NP-vollständig.

In der Literatur finden sich mehrere Ansätze um Power Dominating Set zu lösen, unter Anderem eine Formulierung des Problems als ganzzahliges Programm. In dieser Abschlussarbeit nutzen wir einen Algorithmus, der auf Branch-and-Bound beruht. Dafür leiten wir einige untere und obere Schranken für die optimale Lösung her und verknüpfen das Problem mit einem verwandten Überdeckungsproblem. Des Weiteren stellen wir Strategien für den Branching-Schritt des Algorithmus vor. Die Implementierung unseres Algorithmus ist schneller als das Auswerten des ganzzahligen Programms auf kleinen Graphen, aber deutlich langsamer auf größeren Instanzen.

Contents

1 Introduction

The problem of power dominating set originates from monitoring the status of electric power networks [BBE18 | HHHH02]. This requires measuring the voltage and the phase angle, which can be done by adding *phase measurement units* to the network. Phase measurement units are expensive, so it is of interest to minimize the total amount of such units in a network while still monitoring it fully. The problem can be formalized as a minimization problem on a graph G that represents the electric power network. In this context, we call power measurement units *active vertices* of the graph and capture the notion of monitoring in *observation rules*.

Stated as a graph theory problem, power dominating set looks for an active set of vertices $A \subseteq V$ of an undirected graph $G = (V, E)$ such that all vertices are observed. Observation happens due to two rules: First, the domination rule observes all neighbors of active vertices. In addition to that, the propagation rule can extend the set of observed vertices to those that are the only unobserved neighbor of an observed vertex. The problem is closely related to *dominating set*, which only uses the first rule. Dominating set belongs to the set of 21 problems that were the first to be proven NP-complete by Karp in 1972 [Kar72]. As an extension of dominating set, power dominating set is NP-complete as well [HHHH02].

Power dominating set has been previously studied both from a theoretical perspective, deriving bounds on the solution size for different graph classes [HHHH02 | ZKC06], aswell as proposing algorithms for computation. Binkele-Raible and Fernau present an exponential time algorithm [Bin11], whereas Jovanovich and Voss formulate an integer linear program [JV20]. In this thesis we take a different approach to solving power dominating set by using the branch and bound framework to search the space of possible solutions efficiently. Branch and bound works by alternating between two steps. *Branching* subdivides the search space into two or more subsets, then *bounding* discards parts of the search space that can be proven to not contain a solution. The algorithm can be accelerated by applying reduction rules that simplify the search space. Our algorithm is faster than the reference solution for small graphs, but considerably slower on larger graphs.

We define power dominating set formally in chapter 2 using graph theoretic terms and describe further prerequisites. We also give a concrete example of a power dominating set to illustrate the concept.

Chapter 3 outlines related results for power dominating set as well as other related minimization problems on graphs. In chapter 4 we present our branching algorithm. We discuss the various components that make up a branch and bound algorithm. A core component of such an algorithm are *bounds* for the size of the solution. We derive multiple lower and upper bounds for power dominating set (section 4.2) using a related problem, *subdivided star cover*, which we introduce. Further, we discuss strategies for branching. First, branching on the inclusion or exclusion of a vertex in the set of active vertices (section 4.3), and after that a variant that makes use of a special structure in certain graphs to discard parts of the search space early (section 4.4).

We conclude with an evaluation of our algorithm on graphs from real world data in chapter 5. We determine which strategies for branching are more useful on average and determine groups of bounds that can be sensibly used in conjunction with each other. Lastly, we discuss the performance of our algorithm.

2 Preliminaries

We state power dominating set as a graph problem. A graph G is a structure consisting of vertices V and edges E. For *undirected graphs*, each edge $e \in E$ is a set of two vertices in V, that is $E \subseteq \binom{V}{2}$, whereas edges in *directed graphs* are a 2-tuple of vertices, so $E \subseteq V \times V$. In both cases we denote the number of vertices by n and the number of edges by m. The degree $\deg(v)$ of a vertex $v \in V$ is the number of edges that contain v, so for undirected graphs we have $\deg(v) = |\{e \in E \mid v \in e\}|$. For directed graphs we differentiate between the indegree $\deg_-(v) = |\{u \in V \mid (u, v) \in E\}|$ and the outdegree $\deg_+(v) = |\{w \in V \mid (v, w) \in E\}|$. Additionally we use $\deg(v) = \deg_-(v) + \deg_+(v)$ as the sum of the in- and outdegree for directed graphs. The largest degree in a graph is $d_{\max} = \max_V \deg(v)$.

We also define neighborhoods for vertices of undirected graphs. The open neighborhood $N(v)$ is the set of vertices that share an edge with v, that is $N(v) = \{w \in V \mid \{v, w\} \in E\}$. The closed neighborhood $N[v] = \{v\} \cup N(v)$ additionally includes the vertex v itself. We extend the definitions of neighborhoods to sets, meaning for a set of vertices $S \subseteq V$, $N(S) = \bigcup_{s \in S} N(s)$ and $N[S] = \bigcup_{s \in S} N[S]$. The notation of $\mathcal{P}(S)$ is used to describe the power set of an arbitrary set S.

2.1 Power Dominating Set

2.1.1 Definition

Power dominating set is a minimization problem that asks for the smallest set of vertices in a graph that *observe* all vertices. Any set S that has this property is called a *power dominating set.* More generally, a set of *active vertices* $A \subseteq V$ observes some set $O_A \subseteq V$ of vertices. Vertices are observed iteratively according to two rules [BG23]. First the *domination rule* determines an initial set of observed vertices:

Domination Rule All active vertices are observed. Further, every vertex w that is a neighbor of an active vertex v is observed as well, that is $N[A] \subseteq O_A$. We say v *observes* w.

Additional vertices are observed by applying the propagation rule. We consider only a subset of vertices $P \subseteq V$, called *propagating vertices* to be capable of observing further vertices according to this rule.

Propagation Rule If a propagating vertex $v \in P$ is observed and has only one unobserved neighbor w, then v observes w.

The propagation rule is then applied repeatedly until no additional vertices are observed. If the set of observed vertices O_A matches the entire vertex set, then A is a *power dominating set.* In general, there can be many different power dominating sets S for a given graph. In

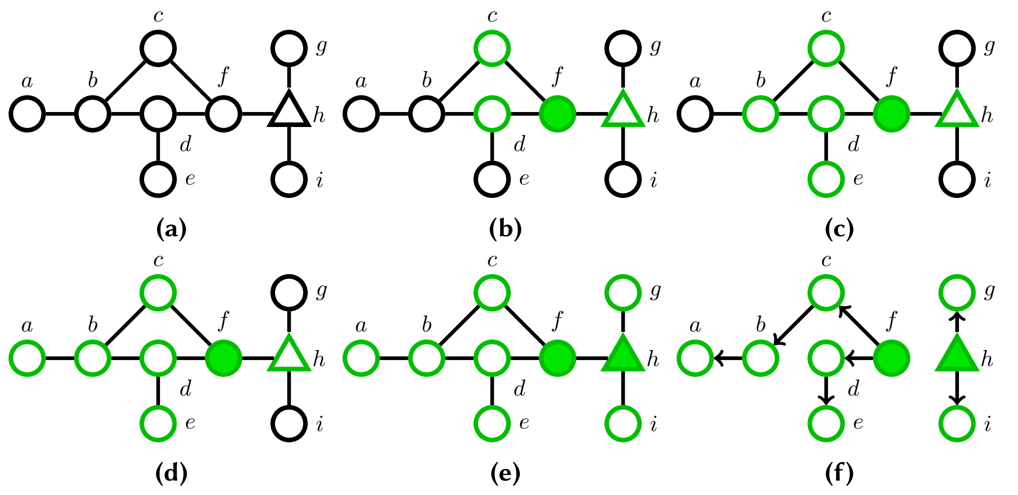


Figure 2.1: Demonstration of the observation rules

particular, the set of all vertices is such a set. We are, however, interested a power dominating set containing as few vertices as possible. We denote a mimnimum power dominating set S by S^* and call $\gamma_P = |S^*|$ the *power domination number.*

If we only consider the domination rule, the problem is called *dominating set*, one of Karp's 21 NP-complete problems. Both problems are equivalent for the special case of graphs without propagating vertices. Therefore any algorithm that can solve power dominating set can also solve dominating set. It follows immediately that power dominating set is NP-complete as well.

To represent partial solutions we partition the vertex set into three disjoint sets. The set of *active* vertices A contains all vertices which have been chosen to be part of a power dominating set. *Inactive* vertices I are vertices which we have ruled out from being part of the power dominating set, and all remaining vertices $B = V \setminus (A \cup I)$ are called *blank* vertices.

2.1.2 Example

To demonstrate the observation rules let us consider the example graph shown in figure 2.1a. Propagating vertices are shown as circles and non-propagating vertices are shown as triangles. Vertices filled in green are active, and vertices with a green outline are observed. First, we choose f as an active vertex. Due to the domination rule, all neighbors of f, namely c, d and h become observed (fig. 2.1b). Next, we check which vertices can observe their neighbors due to the propagation rule. d cannot observe a neighbor because it has more than one unobserved neighbors. h is non-propagating, so it cannot observe any neighbors either. c has only one unobserved neighbor, b, which it therefore observes (fig. 2.1c). Now d has only one unobserved neighbor, so it can now observe e. Additionally a is observed by b (fig. 2.1d).

Now no more vertices can be observed, so in order to find a power dominating set for the graph we have to turn another vertex active. If we want a small power dominating set, choosing i or g is less useful because h will not propagate the observation status. We therefore choose h, which then observes both i and g due to the domination rule (fig. 2.1e). Now $\{f, h\}$ form a power dominating set, and even a minimal power dominating set. It is however not unique, in fact we could have chosen b, c or d instead of f and still would observe all vertices.

2.1.3 Observation Graph

The observation rules imply a directed graph on the vertices V of the input graph. For a set of active vertices A, the *observation graph* $G_A = (V, E_A)$ contains an edge (v, w) if and only if v observes w according to either observation rule. The edge set E_A is in some sense a subset of the edges E of the input graph if we consider the undirected edges $\{v, w\}$ of G to be equivalent to two directed edges (v, w) and (w, v).

Which edges of E are part of E_A also depends on the order in which the rules are applied to the vertices. We assume implicitly that the rules are applied according too some arbitrary but fixed ordering $\leq_V$, and by extension define the observation graph. However, the precise ordering does not affect which vertices are observed after exhaustive application of the observation rules. The observation graph for the graph in figure 2.1a is shown in figure 2.1f.

We also define the *observation neighborhood* $N_A[v]$ for blank vertices $v \in B$ and a set of active vertices A. The observation neighborhood $N_A[v]$ contains precisely those vertices that would become observed by setting v to active, or formally $N_A[v] = O_{A\cup\{v\}} \setminus O_A$.

A graph class that is related to power dominating set are *subdivided stars* which we want to define here. In fact we will show in section 4.2.1.1 that for a power dominating set, the weakly connected components of the observation graph are subdivided stars.

A *subdivided star* is a directed tree with a root vertex c called *center* with $\deg_-(c) = 0$. Edges are oriented away from c, i.e. all other vertices $v \neq c$ have $\deg_-(v) = 1$. Additionally, non-center vertices have outdegree $\deg_+(v) \leq 1$ if they are propagating vertices and $\deg_+(v) = 0$ if $v \notin P$.

Subdivided stars can be seen equivalently as a set of directed paths that are joined at the center vertex and share no other vertices. We discuss the names that are used across literature for this graph class in section 3.4.

2.1.4 Subproblems

A widely used tool for simplifying a complex problem is to divide it into multiple subproblems, solve each on its own and then reassemble them to get a solution for the original problem. This can be trivially applied to power dominating set by finding power dominating sets for each connected component of a graph G and then combining the solutions. This yields a correct result because a vertex can only be observed by vertices in the same connected component of G. The power domination number for a graph is therefore the sum of the power domination numbers of its components.

We can improve this by considering *fully observed vertices.* A vertex is considered fully observed if and only if all vertices in its neighborhood are observed, that is for some set of active vertices A we have $N[v] \subseteq O_A$. Fully observed vertices can not propagate the observation status from neighbors. So if we have a set of fully observed vertices that form a vertex seperator, they act as a boundary across which the observation status can not be propagated. Therefore we can divide G at such seperators, generating multiple subproblems. This is especially useful if many vertices have already been assigned to A.

2.1.5 Reduction Rules

For a partial solution with active vertices A and inactive vertices I, we may be able to infer that other blank vertices must necessarily be active, or conversely can never be part of a minimal power dominating set. Consider for example an isolated vertex, that is a vertex of degree zero. The only way this vertex can be observed is if it is itself active. We call such rules that simplify the structure of the graph whithout discarding solutions *reduction rules*. We use the reduction rules that have been presented by Bläsius and Göttlicher [BG23] to speed up our algorithm. We will take a closer look at one more example here.

Consider a blank vertex $v \in B$ of degree one, also called a *leaf*. It has one neighbor, $w \in B$ which we assume is also blank. Choosing v to be active never yields a smaller solution than choosing w to be active instead. This is because the observation neighborhood of w contains that of v, ie. $N_A[v] \subseteq N_A[w]$. We can therefore set v to be inactive. Moreover, if $w \notin P$ is non-propagating, we need to set w active for v to be observed. Conversely, if $w \in P$, we can equivalently set w to be non-propagating and delete v.

It is noteworthy that these rules do not just reduce the size of the input graph or lower the number of blank vertices but also often generate non-propagating vertices. This is the case even for input graphs that only contain propagating vertices. Graphs that contain many non-propagating vertices have more constrained observation neighborhoods, which we make use of for deriving lower bounds.

2.2 Branch and Bound

Branch and bound refers to an algorithmic framework for finding solutions to optimization problems [MJSS16]. Algorithms based on branch and bound search through some space X of possible solutions trying to find an optimal solution $x^* \in X$ that either minimizes or maximizes some objective function $f : X \to \mathbb{R}$. For the sake of readability we assume we want to solve a minimization problem.

The search space can be organized in a search tree T which the algorithm explores. Starting at a root node we iterate over T in two main steps. The root node contains all possible solutions. In the context of power dominating set this means that all vertices of the graph are blank. (Note that we always use *vertex* for the power dominating set instance and *node* for the branch and bound search tree to differentiate between the two graphs). Each iteration has two main steps. First, the *branching step* generates new child nodes by partitioning the search space at a node x into subproblems $x_1, \ldots, x_k$. If a node still admits an optimal solution then this solution must still be possible in at least one child node. This is always the case if the child nodes cover the entire search space of node x together. For power dominating set, this could mean picking a blank vertex v, and generating two subproblems, one with $v \in A$ and the other with $v \in I$ but is generally not the only possibility. The subproblems are then inserted in T as child nodes to x. By repeating this step we would eventually generate every possible solution (assuming the search space is finite), one of which would minimize f.

To limit the size of the search tree, and by extension, speed up the searching process, we follow up each branching step with a *bounding step*. In this step we want to *prune* subtrees for which we can show that no node can be optimal. We do this by keeping track of the best feasible solution y we have found so far. If $f(y) \leq f(x_i)$ for all nodes x_i in the subtree of node x, we can prune that subtree. To do this, a lower bound f' for f is needed, so $f'(z) \leq f(z)$ must be true for all $z \in X$. We then need to only verify that $f(y) \leq f'(x)$ for the single node x to prune its subtree. Once no nodes remain unexplored, we know that y is the optimal solution.

Even though the search space is organized in a tree, we do not need to explicitly construct T. Instead it is sufficient to keep the unexplored nodes in a priority queue, that has its elements sorted by lower bound in ascending order, corresponding to searching T depth-first.

3 Related Work

3.1 Power Dominating Set

The problem of efficiently placing measurement units in an electric power system formalized as a graph problem has first been stated by Haynes et al. [HHHH02]. Introducing the power dominating set problem as an extension of dominating set, the authors show that any dominating set is always a power dominating set and observe that as a consequence, the power domination number γ_P of a graph is always less than or equal to the domination number γ. They further show that the problem remains NP-complete, even if the graphs are restricted to two common graph classes, namely bipartite and chordal graphs. For trees the problem is easier, in fact a power dominating set for trees can be found in linear time using an algorithm that partitions the input graph into subdivided stars.

An exponential time algorithm for solving power dominating set on general graphs is given by Brueni and Heath [BH05]. It relies on the result that on graphs with at least three vertices, the power domination number is bounded by $\gamma_p \leq \lceil \frac{n}{3} \rceil$. The resulting algorithm iterates then over a set of possible candidates in $O^*(1.89^n)$ time.

This asymptotic bound can be improved by a branching algorithm [Bin11] which requires $O^*(1.7548^n)$ time on general graphs. Each branching step makes the decision to either include or exclude a specific vertex in the power dominating set. The generated tentative solutions are stored using a specific data structure the authors introduce, called *reference search trees*. The algorithm also makes use of reduction rules that simplify the input graph without changing the optimal solution.

A different approach to solving power dominating set is taken by Jovannovic and Voss [JV20]. The authors transform the input graph and observation rules into an integer linear program which then can be solved using a generic solver. This formulation however considers all vertices to be propagating and has been extended to include non-propagating vertices by Bläsius and Göttlicher [BG23].

More general bounds on γ_P for specific graph classes are summarized in a survey by Dorbec [Dor20]. While some of these bounds are tight, they are not very useful for use in a branching algorithm as the input graphs in general do not match any of these graph classes.

In branch and bound algorithms, monotone properties of the optimization problem can be exploited to reduce the size of the search space. As such it would be useful to show that γ_P is monotone with respect to some graph operation. Howerever, Dorbec et al. [DVV16] show that this is not the case for three common local operations. Vertex removal, edge removal and edge contraction can each increase and decrease the power domination number.

3.2 Hitting Set

Hitting set can be expressed as a covering problem on hypergraphs. Given a hypergraph G with vertices V and a set of hyperedges $E \subseteq \mathcal{P}(V)$, we consider a subset of vertices $H \subseteq V$ a *hitting set* if every hyperedge contains at least one vertex in H. This can be formally stated as follows: $H \subseteq V$ is a hitting set if and only if $\forall e \in E : H \cap e \neq \emptyset$. The hitting set problem then asks for the smallest hitting set for a given hypergraph.

The problem can be restated using just a family of sets and is in this context known as *set cover*. It is one of the original 21 problems proven by Karp to be NP-complete [Kar72].

A survey of algorithms [CTF00] for solving hitting set both exactly and heuristically show that most solvers rely on a linear program formulation of the problem. Conversely, Bläsius et al. [BFSW] use a branch and bound algorithm to find a minimal hitting set. They introduce a multiple lower bounds as well as an upper bound, some of which can be adapted for power dominating set.

The simplest lower bound, *max-degree* is based on the degrees of the vertices in a hitting set. The degree $\deg(v)$ of a vertex $v \in V$ in a hypergraph is the count of hyperedges that contain v. Each vertex is contained in at most $d_{\max} = \max_V \deg(v)$ hyperedges. Because every hyperedge needs to contain at least one vertex in a hitting set H, the smallest hitting set contains at least $\left\lceil \frac{|E|}{d_{\max}} \right\rceil$ vertices.

This bound can be improved by observing that instead of $d_{\max}$, each degree can only be chosen as often as it appears in G. Thus, given a list of vertex degrees $d_1, \ldots, d_{|V|}$ sorted in descending order, we can take the smallest number k for which the sum over the largest k degrees $\sum_{i=0}^{k} d_i$ exceeds the number of hyperedges $|E|$. Then k is a lower bound for the smallest hitting set, called *sum-degree*.

We can adapt both lower bounds for finding a smallest power dominating set in a graph. In a similar way that every vertex in a hitting set covers hyperedges according to its degree, a vertex in a power dominating set is limited by its degree in how many non-propagating vertices it can observe.

The authors introduce two further bounds that could in theory be transferred to power dominating set, but would only yield trivial bounds: *Efficiency bound* charges each edge in the hypergraph by a cost of $1/\deg(v)$ for all contained vertices, and bounds the size of a hitting set by the sum of the minimal charges for each edge. *Packing bound* considers a set of pairwise disjoint hyperedges P. The smallest hitting set then is at least as large as P. A correct efficiency bound for power dominating set would require a very small charge for each vertex, namely the inverse of the size of the respective connected component, because without further structural insights the entire component could be observed by just a single vertex. As a result the bound would just equal the number of connected components. Packing bound has similar issues. The observation neighborhood of a vertex could extend over many vertices and depends on which other vertices have already been chosen to be active. Only if two vertices are in different connected components is it straightforward to show that their observation neighborhoods do not intersect.

The sum-degree bound and the packing bound can be combined to get a tighter lower bound. The authors use a greedy algorithm to determine an upper bound for hitting set. By repeatedly selecting the vertex with the highest degree and adding it to H and checking if all hyperedges are covered, a hitting set will eventually be generated. We follow a similar strategy for finding an upper bound for power dominating set.

3.3 Zero Forcing Problems

Zero forcing is described by Bozeman et al. [BBE18] as a coloring game on the vertices V of a graph G. Each vertex has one of two colors, blue and white. The authors denote the set of blue and white vertices by B and W respectively. The vertices change color based on the following rule: If b is blue only one neighbor w of b is white, then the color of w changes to blue. This is denoted by $b \to w$ and called *b forcing w*. The closure of an initial set of blue vertices B is the set of vertices that are blue after the color changing rule is applied exhaustively. A set B for which the closure of B is the entire vertex set V is called a *zero forcing set.* The size of the smallest such set is called the *zero forcing number* $Z(G)$. In essence, zero forcing matches the propagation rule of power dominating set where all vertices are considered to be propagating.

The *chronological list of forces* is a list of vertices that describes the order in which the coloring rule is applied to the vertices. The authors note that sequences of vertices $v_i \to v_{i+1}$ form a path in G. This is corresponds to the observation graphs for observed but inactive vertices in power dominating set. The distinction is due to the domination rule, which causes possibly multiple paths to begin at a single vertex, making the observation neighborhood of a graph a subdivided star and not just a path.

In Bozeman et al. derive various bounds for the zero forcing number and the power domination number $\gamma_P(G)$, as well as an equation that relates the two numbers [BBE18]. Before we can state the equation we need to define *restricted* variants of the two problems. For a given set $X \subseteq V$, we set $B = X$ at the beginning and want to find the smallest zero forcing set under that precondition. We call $Z(G;X)$ the restricted zero forcing number. Similarly for restricted power dominating set, we consider X to be necessarily active vertices and want to find the smallest power dominating set with respect to this precondition and call $\gamma_P(G;X)$ the restricted power domination number. These two figures can then be related by the following tight bound:

$$\left\lceil \frac{Z(G;X)}{d_{\max}(G)} \right\rceil \leq \gamma_P(G;X)$$

From this, the authors go on to derive an integer linear program to determine a minimum restricted power dominating set of a graph. They do this using the concept of a *fort*, a subset $F \neq \emptyset$ of vertices with the property that every $v \in V \setminus F$ has either zero or at least two neighboring vertices in F. Now, any power dominating set S has to intersect the closed neighborhood of all forts F, that is $S \cap N[F] \neq \emptyset$. This can be easily stated as an integer linear program. However it is infeasible to generate every fort, as there can be exponentially many. Instead, a second ILP for constraint generation is given. Based on a partial solution for the first program, it generates an additional fort that leads to a better solution.

The iterative process of solving power dominating set using fort constraints could be used for determining lower bounds in a branch and bound algorithm, as the partial solution is a lower bound on γ_P. Nevertheless, additional care needs to be taken with non-propagating vertices, as the ILP considers all vertices to be propagating.

3.4 Graph Covering

In section 4.2.1.1 we relate the problem of power dominating set to a covering the vertices of a graph with subdivided stars. For general graph covering problems we are given an input graph H as well as a class of graphs $\mathcal{G}$, and a notion of how to cover H with instances of $\mathcal{G}$

[KU16]. Usually one is either concerned with covering the edges of H or the vertices of H. Additionally, some coverings may be considered better than others, for example based on the number of graphs from $\mathcal{G}$ that are needed.

The class of graphs which we call subdivided stars is inconsistently named across literature. Some authors consider the graph resulting from subdividing each edge of a star exactly once to be subdivided stars [Şah22 | BJ18]. Others use an equivalent definition to ours, but call them *spider graphs* [PP16 | HHHH02]. Additionally the term spider graph is also used for stars with exactly one vertex of degree greater than two where all edges have been equally often subdivided [Moh13].

Algorithms for covering the vertices of graphs using similar classes such as trees and stars exist [Eve+03] but to the best of our knowledge no prior research on covering graphs with subdivided stars has been published.

4 The Branching Algorithm

In this chapter, we describe our branching algorithm used to solve power dominating set instances. We outline our algorithm in section 4.1 before discussing the specific components. Then, in section 4.2 we present bounds for γ_P. A group of lower bounds, which are based on the relationship between the observation graph and subdivided stars, are discussed in section 4.2.1.1. After we present three upper bounds based on greedy selection of vertices in section 4.2.3, we offer a comparison of all bounds we present in section 4.2.4. We present strategies for selecting vertices to branch on in section 4.3. Lastly, we conclude the chapter by exploring an alternative *wide branching* strategy in section 4.4.

4.1 Overview

Our algorithm starts out with an input instance for which all vertices are considered to be blank vertices, i.e. $V = B$. During the execution of the algorithm we assign more and more vertices to either the active set A, or the inactive set I. To do this, we keep a list of instances Q, from which we generate new instances that split the search space. Eventually, we reach instances that do not contain any blank vertices. Among these, we want to find one that is a minimum power dominating set.

The algorithm consists of a main loop that iterates over elements of Q, which to begin contains just the input graph. Additionally, we associate a lower bound ℓ and an upper bound u with each instance. Initially we set $\ell = 0$ and $u = |V|$. The main loop terminates when Q is empty. We also keep track of the best known solution G^*, as well as the lowest upper bound u^*.

We now want to take a look at one loop iteration. First, we take a graph G out of Q. In our case Q is implemented as a priority queue and we always remove the element with the smallest lower bound first. To start, we check if the lower bound ℓ of the current graph G is larger than then u^*. If this is the case we can safely skip it, as it can never be an optimal solution. Next we update the lowest upper bound, that is set u^* to $\min u, u^*$. After that we check if the current instance is a valid power dominating set. If it is, we take G as the new best known solution G^*. Note that it is impossible for a valid but suboptimal power dominating set to be found at this step, because for incumbent solutions both bounds as well as the solution size are identical, which would eliminate the suboptimal solution at the start of the loop.

In the next step we need to subdivide the search space. Generally, this means we need to construct a set of new graphs $G_1, \ldots, G_k$ that make further restrictions on G, meaning that their sets of blank vertices each need to be a subset of the blank vertices in G. On the other hand, if the set of active vertices can be extended to a minimum power dominating set, this must still be possible for at least one G_i. In the simplest case we just pick a single blank vertex $v \in B$ and generate two new graphs. The first one contains v as an active vertex whereas in the second one, v is inactive. All other vertices remain as they were in G.

Alternatively we can subdivide the search space in more than two parts. This is called *wide branching* [MJSS16] in general. We present one such scheme for power dominating set specifically in section 4.4. Note that the branching scheme does not need to be the same across the entire algorithm but can instead be mixed and matched.

To simplify the graph we apply the reduction rules presented by Bläsius and Göttlicher [BG23]. We apply these rules exhaustively, meaning until no more rule can be applied, to all graphs $G_1, \dots, G_k$. We also do this in the beginning to the input graph.

Lastly we compute the lower and upper bounds on each new graph before adding them to Q. We repeat the checks on bounds from the beginning of the loop aswell as the update of u^* here. This is not strictly necessary, but keeps the list from getting unnessecarily large and can cause other instances to be discarded earlier.

The runtime of the algorithm strongly depends on the the bounds and how fast we can prune parts of the search space. In the worst case, without considering reduction rules, we would generate all $2^{|V|}$ possible assignments from V to $\{A, I\}$ culminating in exponential runtime.

Algorithm 4.1: A branching algorithm for power dominating set.

```
Input:  A graph $G$ containing only blank vertices.
Output: A power dominating set for $G$ of minimal size.

1  $G.exhaustiveReductions()$
2  $u^* \longleftarrow |V|$
3  $G^* \longleftarrow null$
4  $Q \longleftarrow newQueue()$
5  $Q.insert((G, \ell = 0, u = u^*))$
6  while $Q \neq \emptyset$ do
7      $(G, \ell, u) \longleftarrow Q.getMin()$
8      if $\ell > u^*$ then
9          continue
10     $u^* \longleftarrow \min u, u^*$
11     if $O_A = V$ then
12         $G^* \longleftarrow G$
13     $G_1, \dots, G_k \longleftarrow subdivideSearchSpace(G)$
14     for $j \in \{1, \dots, k\}$ do
15         $G_j.exhaustiveReductions()$
16         $\ell, u \longleftarrow computeBounds(G)$
17         if $\ell < u^*$ then
18             $Q.insert((G_j, \ell, u))$
19             $u^* \longleftarrow \min u, u^*$
20 return $G^*$
```

4.2 Bounds for Power Dominating Set

The efficiency of a branch and bound algorithm strongly depends on the bounds that are used to constrain the search space. In this section we present and analyze a variety of both lower and upper bounds on power dominating set instances. To do this, we first define lower and upper bounds. A lower bound $\ell(G)$ is a function that for all graphs G with partitions of its vertices into active, inactive and blank vertices takes on a value at most as large as the smallest power dominating set possible with the already assigned vertices. Conversely, an upper bound $u(G)$ is a function that is always at least as large as the smallest power dominating set for a given graph. In notation we usually omit the dependence on the graph G for our bounds.

Note that multiple bounds of the same kind can be combined to form a new bound. If $\ell_1, \dots, \ell_k$ are all lower bounds, then $\max(\ell_1, \dots, \ell_k)$ is also a correct lower bound. Similarly, the minimum of a set of upper bounds is an upper bound as well. This is useful because, in general, which bound is best may depend on the graph. If one lower bound ℓ_1 is greater than ℓ_2 independent of the instance, we say ℓ_1 *dominates* ℓ_2. For upper bounds, a dominating bound is always smaller than the dominated bound.

Bounds that are closer to the optimal solution γ_P are generally favorable, as they reduce the space of possible solutions to a larger extent. As a consequence, the power dominating number itself could be considered the best bound, as it is both the largest lower bound aswell as the smallest upper bound. So an additional requirement for a good bound should be the ability to compute it quickly. For this reason, bounds that are dominated by other bounds can still be useful if they can be evaluated faster.

Before we explore more elaborate bounds, we want to give a pair of exemplary bounds based on the number of active vertices. If A is not a power dominating set, we need at least one more vertex to observe the full graph. Thus

$$a_\ell = |A| + \begin{cases} 1 & \text{if} \quad O_A \neq V \\ 0 & \text{else} \end{cases}$$

is a lower bound. Similarly, we can construct an upper bound a_u:

$$a_u = |A| + |B \cap U_A| + \begin{cases} 1 & \text{if} \quad O_A \neq V \\ 0 & \text{else} \end{cases}$$

If it is still possible to find a power dominating set given the set of inactive vertices I, then setting all unobserved blank vertices to active clearly is a solution. In some cases however, too many vertices are inactive, so the instance is infeasible and can be discarded. Both bounds can be computed in $\mathcal{O}(n)$ time by iterating over all vertices.

4.2.1 Star Bounds

4.2.1.1 Star Cover

The set of lower bounds we present next are derived from properties of the observation graph G_A of G. First, we show the relation between the observation graph and the graph class of subdivided stars (see section 2.1.3).

Theorem 4.1: Let $A \subseteq V$ be a set of vertices of a graph G. Then each weakly connected component of the observation graph G_A is either

1. a single vertex of degree zero which is unobserved
2. a subdivided star with exactly one active vertex $c \in A$ which we call *center*

Further, if A is a power dominating set of G, only the second case is possible.

Proof. During application of the observation rules, the theorem holds as an invariant. Before any rule is applied, G_A does not contain any edges, thus the invariant is trivially fulfilled.

Applying the domination rule to a vertex $v \in A$ generates a directed edge from v to every unobserved neighbor $w \in N_G(v) \cap U_A$. Therefore the indegree of v is zero and its outdegree is unconstrained, making it the center. All neighbors w have indegree one and outdegree zero. Thus each observed vertex is part of a subdivided star in G_A and all unobserved vertices are isolated, so the invariant holds.

Every application of the propagation rule requires an observed vertex v with exactly one unobserved neighbor w. The vertex v cannot be active, otherwise it would not have an unobserved neighbor. Because v is observed and not active, it must have outdegree zero. The rule introduces an edge (v, w) in G_A, raising v's outdegree by one and setting the indegree of w to one. Therefore the invariant also holds after application of the propagation rule.

Lastly, if A is a power dominating set, all vertices must be observed, thus eliminating the possibility of the first case. ∎

We can now define the notion of a *subdivided star cover.* Given an undirected graph $G = (V, E)$, we call a directed graph $G' = (V, E')$ on the same vertex set a subdivided star cover of G if two conditions are met. First, G' must be a subgraph of G, that is for every edge $(u, v) \in E'$ an edge $\{u, v\}$ must exist in E. Additionally every weakly connected component of G' must be a subdivided star. We call the number of components the *size* of the subdivided star cover. Furthermore we call the subdivided star cover of minimal size the *subdivided star cover number* and denote it by s.

From Theorem 4.1 immediately follows that the observation graph G_S of a power dominating set S is also a subdivided star cover of G, although not necessarily a minimal one. This is also true for the special case of a minimal power dominating set S^*. Because every subdivided star contains exactly one active vertex we get

$$s \leq |S^*| = \gamma_P,$$

so s is a lower bound for the power dominating number.

To use s in our branch and bound algorithm, we need to be able to compute it. This is difficult, in fact it is straightforward to show that computing s is NP-hard. Consider a graph containing only non-propagating vertices, so $P = \emptyset$. Now every subdivided star only contains edges that start at the center of a star. In a sense we are now interested in finding a non-subdivided star cover. Note that finding such a cover of minimal size is directly equivalent to finding a dominating set as each star corresponds to one vertex in the dominating set. Thus, if we had a polynomial time algorithm for computing s, we could also solve a problem that is NP-complete. Therefore the computation of s is NP-hard as well.

We describe how s can be computed for a given graph exactly in section 4.2.1.3, but for now we want to find a way to compute a lower bound for s itself more efficiently.

4.2.1.2 Lower Bounds for s

We notice that a subdivided star can also be characterized as a set of directed paths that start at a shared center vertex. In the observation graph G_S of a power dominating set S these paths end due to two possible reasons. On the one hand, the last vertex can have no unobserved neighbors that can be observed by the propagation rule. Alternatively, the last vertex can be non-propagating and thus the propagation rule does not apply.

Even though not every path ends at an non-propagating vertex, every such vertex does end a path in G_S unless it is itself a center. We also know that the number of paths emanating from an center v is constrained by the degree of v. The number p of all paths in the observation graph G_S must therefore be smaller than the sum of the degrees of center vertices:

$$p \leq \sum_{v \in S} \deg(v)$$

If the power dominating set S contains q non-propagating vertices, r of which are centers, $q - r \leq p$ must hold, and by extension

$$q - r \leq \sum_{v \in S} \deg(v).$$

We can move r to the other side of the equation and, because $r \leq |S|$, get:

$$q \leq \sum_{v \in S} 1 + \deg(v).$$

By substituting $\deg(v)$ with the largest degree $d_{\max}$, we bound the expression from above and can write the sum as a product:

$$q \leq \sum_{v \in S} 1 + \deg(v) \leq \sum_{v \in S} 1 + d_{\max} = |S| \cdot (1 + d_{\max})$$

After rearranging the equation we get a lower bound for the size of S which we call $\tilde{s}$:

$$\tilde{s} = \frac{|V \setminus P|}{1 + d_{\max}} = \frac{q}{1 + d_{\max}} \leq |S|$$

Unlike s, $\tilde{s}$ is easy to compute. In fact, if the underlying data structure keeps track of the largest degree and number of non-propagating vertices, we can compute it in $\mathcal{O}(1)$ time. Our implementation currently does not do this so we iterate over all vertices to determine $d_{\max}$ and q resulting in an asymptotic runtime of $\mathcal{O}(n)$.

This bound has two major shortcomings which we will address next. First, it does not incorporate any information about vertices that have already been assigned to A. Therefore $\tilde{s}$ will only improve during the branch and bound algorithm if reduction rules simplify the graph. Moreover, its quality strongly depends on how good $d_{\max}$ approximates the degrees of other vertices. In many input graphs, we see a few vertices with fairly high degrees and then many vertices with much lower degrees. We improve this aspect of the bound first.

Taking a step back, we notice that the inequality

$$q \leq \sum_{v \in S} 1 + \deg(v)$$

must hold for any power dominating set S. Even though we do not know which vertices S contains and therefore cannot calculate the sum precisely, each vertex $v \in V$ can only appear in the sum once. Let $v_1, v_2, \ldots, v_n$ be the list of all vertices in V ordered by degree such that $\deg(v_i) \geq \deg(v_{i+1})$ for $i \in \{1, \ldots, n-1\}$. Then picking vertices in this order until the inequality is fulfilled makes for an improved bound. More precisely, we define a lower bound $\bar{s}$ as follows:

$$\bar{s} = \min \left\{ k \;\middle|\; |V \setminus P| \leq \sum_{i=1}^{k} 1 + \deg(v_i) \right\}$$

To compute this we can simply collect all the degrees in a list, sort it, and count how many values we need to sum until the threshold of $|V \setminus P|$ is reached. Asymptotically the time needed to sort the list dominates the runtime of the algorithm resulting in $\mathcal{O}(n \log n)$ time over all.

We can speed up this calculation by observing that even though each vertex can appear only once in the sum, each degree can be present multiple times. As noted earlier, especially for lower values, it is common that many vertices share the same degree. So instead of keeping a list of every vertex, we generate a list D with $d_{\max} + 1$ entries that stores at index d, how many vertices in G have degree d.

After that we iterate over D trying to greedily exceed the threshold of $T = |V \setminus P|$. We keep track of the value of the sum x after choosing the k vertices with the largest degrees. Both x and k are initialized with zero. Starting at the largest degree we first calculate how many vertices of degree d we would need to reach T. This value, denoted by j can be fractional. If there are at least j many vertices of degree d available, then $k + \lceil j \rceil$ vertices are sufficient to reach T and we can return. Otherwise we take all $D[d]$ vertices and add the appropriate value to x and increment k by $D[d]$. We repeat this while decrementing the degree until the threshold is reached.

Algorithm 4.2: Fast computation of $\bar{s}$.

```
  Input:  List of degrees $D$, threshold $T$.
  Output: Number of vertices needed to exceed $T$.

1  $x \longleftarrow 0$
2  $k \longleftarrow 0$
3  for $d \in \{d_{\max}, \dots, 0\}$ do
4      $j \longleftarrow \frac{T-x}{d+1}$
5      if $j \leq D[d]$ then
6          return $k + \lceil j \rceil$
7      else
8          $x \longleftarrow x + D[d] \cdot (d+1)$
9          $k \longleftarrow k + D[d]$
10 return false
```

The list D can be generated by iterating over V and incrementing the relevant entry for each vertex. This takes $\mathcal{O}(n)$ time. Computing the bound then takes $\mathcal{O}(d_{\max})$ time as each loop iteration takes constant time. The largest degree can be at most $n - 1$, thus making the over all runtime $\mathcal{O}(n)$, an improvement over the naive solution.

Next, we want to improve $\tilde{s}$ by taking into account which vertices have already been selected for the power dominating set. The active vertices A imply a set of vertices O_A that is already observed. In addition to the $|A|$ vertices that are already active, we need to cover the unobserved vertices with additional subdivided stars. Restricting the computation to unobserved vertices is too strong, because the optimal solution might require an observed blank vertex to be set to active in order to observe a neighboring unobserved vertex. Therefore we need to take all vertices into account that have a neighborhood that is not completely observed. Denoting the largest degree among these vertices by $d^U_{\max}$, we get an improved bound of

$$\tilde{s}^U = |A| + \frac{|V \setminus (P \cap \{v \in V \mid N_A[v] \subseteq O_A\})|}{1 + d^U_{\max}}$$

following the same construction as for $\tilde{s}$.

It may seem like the observed vertices can have an effect on the unobserved area due to the propagation rule, but this is not an issue. Consideder in a graph with $V = P$ vertices $v_1, \ldots, v_k$ that form a path emanating in O_A from some active vertex v_1. The path ends at v_k because it has at least two unobserved neighbors $u_1, \ldots, u_k$. If a new active vertex w_1 were placed in the unobserved region, generating a path of newly observed vertices $w_1, \ldots, w_k, u_1$ that ends next to v_1's path, the observation graph my extend due to the propagation rule. If v_k has only two unobserved neighbors, then u_2 becomes observed as well, however it depends on the vertex order which subdivided star continues. It is also possible that the subdivided stars are split in a different way and that some edges get reversed. Even though this is ambiguous, the number of subdivided stars and the overall observed region stays the same.

We can easily see that $\tilde{s}^U$ dominates $\tilde{s}$. Every set of subdivided stars corresponding to a bound of $\tilde{s}^U$ is also admissable for $\tilde{s}$ although not the other way around. Therefore, $\tilde{s} \leq \tilde{s}^U$ for every instance of power dominating set. This bound can be computed in $\mathcal{O}(n)$ as well, however a $\mathcal{O}(1)$ computation as for $\tilde{s}$ would require the graph structure to keep track of additional data.

One further improvement to $\tilde{s}$ can be made by recognizing that the observation neighborhoods of any vertex can not extend to different subproblems. We can therefore subdivide the graph into its subproblems and calculate $\tilde{s}$ seperately. Strictly speaking, the subdivided stars can rearrange as described above, but this is no problem for the same reasons. This seems similar to the previous strategy, but is subtly different. For $\tilde{s}^U$, we only consider the set of unobserved vertices as a whole, whereas in this case, we consider each region describing an individual subproblem on its own, but do not care whether the vertices in it are observed or not. Because the subproblems can share some active vertices, we need to determine how much the bound raises the number of active vertices required per subproblem. After that we can add the changes up and add the number of active vertices in the entire graph to get the improved bound which we denote by $\tilde{s}_C$. Following from the construction, we see that $\tilde{s}_C$ dominates $\tilde{s}$. Computing the bounds for each subproblem can be done in linear time in the number of its vertices, but computing the subproblems in the first place takes considerably more time.

Of course we can also combine the three strategies for improving $\tilde{s}$ to get better bounds. In addition to $\tilde{s}, \bar{s}$ and $\tilde{s}^U$ we also compute $\bar{s}^U$, where we take the individual degrees of vertices in unobserved regions into account, aswell as $\bar{s}^U_C$ where we combine all three strategies. We do not use the simple $\tilde{s}$ bound in conjunction with more costly operations because computing $\bar{s}$ instead takes a negligable additional amount of time.

4.2.1.3 Exact Formulation

Next we want to calculate the subdivided star cover number s for a graph $G = (V, E)$ precisely. On the one hand it is a better bound than the approximations discussed in the previous section, so we can prune larger parts of the search space in the branching algorithm, but it also serves as a point of reference to assess the quality of the other bounds. As previously discussed, computing s is NP-hard in general. For this reason we choose a common approach for solving such problems by defining an integer linear program which can be solved using a generic algorithm.

To do this, we model a valid subdivided star cover using three kinds of decision variables. First, we introduce a variable $c_i \in \{0, 1\}$ for every vertex in V, indicating whether the corresponding vertex is a center of a subdivided star. Because every vertex in V needs to be connected to a center vertex, we introduce parent pointers $p_{ij} \in \{0, 1\}$ that form paths towards center vertices. We add p_{ij} and p_{ji} for every undirected edge $(i, j) \in E$, because the parent pointers can be directed in either way. Lastly we add distance variable d_i for every vertex that can take on any natural number. It indicates how often we need to follow a parent pointer until we reach a center. Additionally we use parameter $z_i \in \{0, 1\}$, that is one if and only if the corresponding vertex is propagating.

This construction already guarantees that the subdivided star cover is a subgraph of G, because any edge not in E does not have a corresponding variable. Next we describe the constraints that force every connected component to be a subdivided star.

Distance start Every center has distance zero, so $c_i = 1 \implies d_i = 0$. Note that no distances larger than $|V|$ can occur. Therefore we can add the following constraint for every vertext $v \in V$:

$$d_i \leq |V| (1 - c_i)$$

Here we multiply a binary variable with an upper bound on the range of possible values, thereby conditionally constraining a second variable. We use this technique in many constraints to translate an implication into a linear inequality.

Distance increment To avoid parent pointers creating circles, we keep track of the distance to the center vertex. We require for every edge p_{ij} that $d_i \geq d_j + 1$. This means that no circles of vertices with equal distances are possible It can be expressed by the following constraint:

$$d_i \geq d_j + 1 - |V| (1 - p_{ij})$$

Degree constraints The next three constraints guarantee that the degrees of each vertex match the requirements of subdivided stars. First, we make sure that every vertex that has children must also have a parent, or be a center. This is guaranteed in part by adding

$$\sum_i p_{ij} \leq \sum_j p_{ji} + |V| c_i.$$

Here, the left side determines how many children the vertex j has and the sum on the right side determines j's number of parents. We also need to constrain the number of parents per vertex to at most one:

$$\sum_j p_{ij} \leq 1$$

The next constraint allows center vertices to have arbitrarily many children, non-center but propagating vertices to have one child and vertices that are neither centers nor propagating to have no children at all:

$$\sum_i p_{ij} \leq z_i + |V| c_j$$

Cover Constraint Every vertex needs to be either a center or have a parent pointer so we can find its corresponding center vertex. So for every vertex we must have

$$c_i + \sum_j p_{ij} \geq 1.$$

Active and inactive vertices We want to take account which vertices have previously been chosen to be active or inactive. Vertices in A must be centers, whereas vertices in I can never be centers. Therefore we add the constraints $c_i \geq 1$ for active vertices and $c_i \leq 0$ for inactive vertices.

Optimization Now, any values of c_i, d_i and p_{ij} that fulfill these conditions describe a valid subdivided star cover. To get the smallest subdivided star cover, we minimize the sum of c_i:

$$s = \min \sum_i c_i \quad \text{such that the above conditions are met}$$

The integer linear program can now be solved by a generic solver. Our implementation uses gurobi.

4.2.2 Relaxed Linear Program

Power dominating set has also been solved by formulating it as an integer linear program [JV20]. Solving an integer linear programs is NP-hard in general, however in practice efficient computation is possible for many instances. If the integrality constraint is relaxed, we get a linear program which can be solved more efficiently, both in asymptotic running time and in practice. Because power dominating set is a minimization problem, relaxing the integer linear program enlarges the space of possible solutions towards lower values. Thus, we can relax the program presented by Jovanovic and Voss [JV20] and extended by Göttlicher [BG23] and use the solution as a lower bound. The result can be fractional because we do no longer require the decision variables to be integers. In this case we can round up to the next integer because γ_P is always an integer and rounding down would no longer be a solution for the linear program. We denote this value by r.

Further improvements can be made by solving the relaxed program for each subproblem individually. To do this we compute all subproblems and compute r for each of then. We then add up the new active vertices for each subproblem aswell as the number of vertices that were already active in the entire graph. We call this bound r_C.

4.2.3 Upper Bounds

In addition to lower bounds, we can derive upper bounds for γ_P as well. Upper bounds are useful because they can be used to prune instances from the search tree that can never be optimal. We use a simple greedy approach to find a power dominating set with already chosen active vertices A and inactive vertices I. To do this, we iteratively pick a blank vertex v and add it to the set of active vertices. After that we determine which vertices are now additionally observed and repeat until the entire graph is observed, making A a power dominating set. Note that if inactive vertices have been chosen such that even $A \cup B$ is not a power dominating set, we return a number that is larger than any power dominating set of G can be. We also apply reduction rules exhaustively at the start of every iteration to simplify the graph. This solution is not optimal in general and my contain more than γ_P vertices, thus making it an upper bound.

The precise algorithm depends on the strategy used to pick new vertices v. In section 4.3 we present three such strategies. We define an upper bound for each strategy: Choosing the next vertex by largest degree, we get g_d as an upper bound. If we use the largest observation neighborhood instead, we get g_o. Lastly, g_p describes the upper bound determined by picking vertices in proximity to already determined vertices.

Algorithm 4.3: Greedy upper bound for power dominating set.

```
Input:  Graph $G$ with active vertices $A$, blank vertices $B$ and inactive vertices $I$.
Output: Upper bound for $\gamma_P$
1 while $O_A \neq V$ do
2     $G.exhaustiveReductions()$
3     if $B = \emptyset$ then
4         return $|V| + 1$
5     $v \longleftarrow G.pickBlankVertex(B)$
6     $A \longleftarrow A \cup \{v\}$
7     $B \longleftarrow B \setminus \{v\}$
8 return $|A|$
```

4.2.4 Comparison

To conclude this section, we want to compare the bounds we have presented. All bounds are shown in figure 4.1. Here, an arrow from a to b means that a is at least as large as b for any graph G and partition of vertices into active, blank and inactive.

Comparing the three greedy upper bounds g_d, g_o, g_p from a theoretical perspective is difficult because their precise behaviour depends on the reduction rules that are applied as an in between step. Each of them however dominates a_u, because a_u selects every unobserved blank vertex whereas the greedy bounds might select fewer blank vertices.

The subdivided star cover number s dominates a_l because if the active vertices do not observe the entire graph we get $a_l = |A| + 1$. Because of the cover constraint in the ILP formulation of s, we get $s \geq |A| + 1$ and thus $s \geq a_l$. All the lower bounds for s we described in section 4.2.1.2 rely on non-propagating vertices, so if all blank vertices are propagating, they can be a worse bound than a_l. In practice this is rare, and we expect the lower bounds for s to be greater than a_l, but it shows that neither bound dominates the other. Of course the lower bounds for s dominate each other according to their construction. The diagram also includes the combinations of improvements for $\tilde{s}$ which we do not implement.

We can combine bounds that do not dominate each other (or where we do not know if they dominate each other) to get better bounds. The combined lower bound is $c_l = \max\{r_C, s\}$ and the combined upper bound is $c_u = \min\{g_o, g_l, g_d\}$. Nevertheless, other combinations of bounds can still be useful in the bounding step of the algorithm because they might be faster to compute. In chapter 5 we discuss which bounds are useful based on their real world performance.

4.3 Vertex Selection Strategies

In the branching step of the algorithm we need to split the search space into at least two parts. The approach we follow for the most part, is to pick a blank vertex $v \in B$ and generate two new instances, one in which v is active, and another where v is inactive. These two instances together cover the entire search space, so no possible solutions are discarded. On the other hand, they are also disjoint, which is not strictly necessary but avoids duplicate computations. The order in which blank vertices are picked has an effect on how quickly parts of the search space can be pruned. Therefore we want to eliminate as many subtrees that do not contain an

$$\begin{tikzcd}
 & a_u \arrow[dl] \arrow[d] \arrow[dr] & \\
g_d \arrow[dr] & g_o \arrow[d] & g_p \arrow[dl] \\
 & \gamma_P \arrow[dl] \arrow[d] & \\
r_C \arrow[d] & s \arrow[d] \arrow[dr] & \\
r & \bar{s}^U_C \arrow[dl] \arrow[dd] \arrow[dr] & a_\ell \\
\bar{s}_C \arrow[ddd] \arrow[ddr] & & \bar{s}^U \arrow[ddl] \arrow[ddd] \\
 & \tilde{s}^U_C \arrow[ddl] \arrow[ddr] & \\
 & \bar{s} \arrow[dd] & \\
\tilde{s}_C \arrow[dr] & & \tilde{s}^U \arrow[dl] \\
 & \tilde{s} &
\end{tikzcd}$$

Figure 4.1: Relations between lower and upper bounds:
a_u is an upper bound based on the number of active vertices.
g_d, g_o and g_p are upper bounds based on greedily adding active vertices in order of largest degree, largest observation neighborhood and proximity to non-blank vertices respectively.
r is a lower bound found by relaxing an ILP for γ_P; r_C takes individual subproblems into account.
a_ℓ is a lower bound based on the number of active vertices.
s is the subdivided star cover number.
$\tilde{s}$ is a lower bound for s using the largest degree as an approximation, $\bar{s}$ uses the individual degrees.
A superscript ^{U} indicates restriction to unobserved regions and a subscript $_C$ computes the bound for each subproblem seperately.

optimal solution as quickly as possible. In this section we present three strategies for picking the next vertex to branch on, given some previous assignments of some vertices into active and inactive sets.

Largest Degree The motivation for this strategy is simple: vertices with more neighbors have a stronger influence on the graph. By picking the blank vertex v_d which has the largest degree, we might expect to be able to apply more reduction rules, thereby simplifying the graph more. Finding v_d (or a blank vertex with maximal degree, if there is no unique one) by iterating over all vertices takes $\mathcal{O}(n)$ time, making this strategy asymptotically the fastest.

Largest Observation Neighborhood Perhaps a more precise definition of the influence a vertex v on the graph is captured by the notion of the observation neighborhood $N_A[v]$. Because the observation neighborhood contains exactly the vertices that become observed by adding v to the set of active vertices A, it describes the range of vertices that are affected by branching on v well. We denote the blank vertex that has the largest observation neighborhood by v_o. We can find v_o by determining the observation neighborhood for each vertex, which in turn can be done by applying the observation rules exhaustively. When calculating the observation neighborhood, we need to look at each edge at most two times, so the asymptotic runtime for determining v_o is in $\mathcal{O}(n \cdot m)$.

Proximity For this strategy we take a different approach. We notice that reduction rules can be applied the most if multiple vertices that have been assigned to A and I are close together. Using the other two strategies, we may find vertices that have a large effect on the graph, but if the vertices that are picked in consecutive branching steps are far apart from each other, they can not have a combined effect. Instead we want to choose vertices close to other vertices that have been either picked in a previous step, or were assigned to A or I due to reduction rules. To be more precise, the picked vertex v_p is one that minimizes the proportion of blank vertices among its neighbors:

$$v_p = \operatorname*{argmin}_{v \in B} \frac{B \cap N[v]}{N[v]}$$

Finding v_p by calculating this ratio for every vertex takes $\mathcal{O}(n \cdot m)$ time.

4.4 Branching on Short Paths

So far, the method we chose to subdivide the search space for the branching algorithm relied on picking a single blank vertex and assigning it to the active or inactive set. This splits the search space into two disjoint parts, generating a binary search tree. If we consider more generally a set of k blank vertices, we need to generate 2^k instances, each representing one combination of assigning vertices A or I. This happens automatically during the branch and bound process, although not necessarily in immediate succession.

In this section we want to describe how we can construct these combinations explicitely for a specific structure in G, such that some combinations never need to be constructed because they can never be part of a minimal power dominating set. This structure, wich we call *short paths*, consist out of three blank vertices u, v, w, which need to fullfill the following conditions: As the name suggests, the vertices need to form a path, that is (u, v) and (v, w) need to be edges of the graph. We also require the middle vertex v to have no other neighbors than u and w. Additionally, to reduce the number of possible instances, we want the outer vertices to be non-propagating and the middle vertex to be propagating, that is $u, w \notin P$ and $v \in P$. We illustrate a short path in figure 4.2.

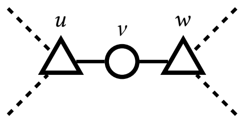


Figure 4.2: Three vertices forming a short path

There can be $2^3 = 8$ possible assignments of the vertices u, v, w to A or I which we will explore next. First, consider the case where all three vertices are inactive. Now, the middle vertex v can never be observed, because the three vertices which could observe v through the domination rule are inactive, and other vertices that could observe v through the propagation rule can never reach v because both u and w are non-propagating. Therefore this case can never be a power dominating set and does not need to be constructed.

The reverse situation where all three vertices are active can be a part of a power dominating set, but never a minimal power dominating set. This is because v does not need to be active to be observed, as it is observed by either u or w. Therefore we could construct a smaller power dominating set which has $v \in I$. As such this case can also be discarded.

Next, consider the three cases where exactly one of u, v, w are active (fig. 4.3). These cases can clearly be part of a power dominating set, as the active vertex can observe both inactive vertices. There is also no reason why for a general graph instance this could never be optimal. It is possible that the active vertices outside of the short path can not observe the inactive vertices within, therefore making the active vertex necessary. As a result, we need to add these three instances to the search space.

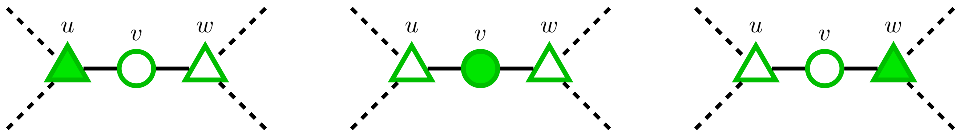


Figure 4.3: Three necessary candidates for short paths

Among the three cases with exactly two active vertices, let us look at the case where $u, w \in A$ and $v \in I$. Again, this can be part of a power dominating set because v can be observed by the outer vertices. This solution can also be optimal, because u and w may be needed to observe vertices outside of the short path. As an example this is the case when both u and w each have two additional neighbors each with degree one as shown in figure 4.4. Here, any other pair of active vertices is not a power dominating set, so the assignment is optimal, and as a consequence, we need to construct this instance.

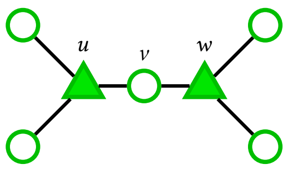


Figure 4.4: A minimal power dominating set for a short path with two active vertices

Of the two remaining, symmetrical cases, consider the one where $u, v \in A$ and $w \in I$. As above, this can be part of a dominating set because v can observe w. It can also be part of an optimal solution if u is needed to observe vertices outside of the short path. However, in this case, choosing w to be active instead of v also generates a power dominating set of the same, minimal size. Because we are only interested in finding *one* optimal solution, we never need to construct this instance, and know that if it would allow for a minimal solution, we account for it in the previous case. The same argument holds for the mirrored case where $v, w \in A$ and $u \in I$, so we do not construct this instance either.

We can further eliminate a case if one of the outer vertices u and w are of degree two. If $\deg(u) = 2$, we get a path of four vertices t, u, v, w. Now the case where $u, w \in A$ and $v \in I$ can also be discarded, because we can pick t instead of u as an active vertex. This does not make the set of observed vertices smaller, and therefore makes the case $w \in A$, $u, v \in I$ at least as good. The case of $\deg(w) = 2$ follows analogous.

Through this construction, we reduce the number of instances in the subdivided search space from eight to four or in some cases even three. To find short paths in the graph, we search the vertex set for possible middle vertices v. For each vertex, we can assess if it is part of a short path by checking the size of its neighborhood in constant time. We further check if the neighboring vertices are both blank and non-propagating if $\deg(v) = 2$, because only then these checks are possible in constant time as well. Therefore finding a short path, or verifying that none exist is possible in $\mathcal{O}(n)$. If we cannot find any, we use the regular strategy of branching on the inclusion of a single vertex instead.

5 Evaluation

To evaluate the algorithm, we implemented it in C++20 and compiled it with GCC v12.0.1 using the `-O3` optimization flag. The program is executed on a machine equipped with a 48-core AMD K10 CPU and 256 GB RAM running Ubuntu 22.04 LTS. We solve linear programs using gurobi v9.5.2.

For testing we mostly use the same graphs as Jovanovic and Voss [JV20], however we evaluate our algorithm on individual subproblems instead of the entire graphs. We do this because it gives us a better idea of how parameters of the graph change the performance of the branching algorithm. A very large graph might consist out of many small subproblems that can be easily solved, so it makes more sense to compare these small subproblems with small input graphs. A different, similarly large graph may contain just a single subproblem that is considerably harder to solve. Before the subproblems are computed we apply the reduction rules exhaustively. Many instances, especially the smaller ones contain only propagating vertices, in others up to 42% of vertices are non-propagating.

We first evaluate the strategies used for branching (section 5.1). After that we take a closer look at the performance and effectiveness of our bounds (section 5.2) and determine which groups of bounds are useful to use in conjunction with each other. We also discuss how well our lower bounds for the subdivided star cover number s approximate s. Last, we evaluate the performance of our algorithm with different sets of bounds on select instances in section 5.3.

5.1 Strategies

We first want to evaluate which of the three strategies for picking vertices to branch on is the most efficient. We combine this with the option of branching on short paths, if possible. To determine the efficiency, we compare how many nodes of the search tree the algorithm has to explore before it terminates. Every node that is taken from the priority queue is counted as explored. A strategy that needs fewer explored nodes is thus more efficient for a given instance. To aggregate the result over all subproblems, we take the strategy of branching at the largest degree as a baseline. Then we calculate the ratio of how much more nodes different strategies explore in comparison and average over all instances. We also measure the relative duration of determining the new instances.

The results in table 5.1 show that picking the largest degree without branching on short paths is on average the most efficient strategy. The other two strategies need to explore roughly one third more nodes before terminating. Branching on paths makes the respective strategies slightly less efficient for both the largest degree and the largest observation neighborhood. For the proximity strategy however, we get a significant improvement, getting results closer to the largest degree. The runtimes are fairly similar, with largest degree being the fastest still. Branching on paths incurs a small performance hit except for largest observation neighborhood where it improves performance slightly.

Table 5.1: Number of explored nodes and runtime relative to the first line

Strategy	Branch on paths	explored nodes	runtime
Largest Degree	No	1.00	1.00
	Yes	1.01	1.04
Largest Observation	No	1.28	1.16
Neighborhood	Yes	1.29	1.06
Proximity	No	1.34	1.03
	Yes	1.06	1.11

In general, we observe that most graphs have no or very few short paths that can be used for branching. On some large graphs, we can branch on short paths about half the time for the first one hundred explored nodes. After that the ratio is much smaller. Even where branching on paths is possible more often than on most other graphs, it does not have a significant effect on the algorithm.

Averaging over subproblems somewhat obscures the difference in efficiency and runtime on individual instances. However there seems to be no strong correlation between the number of vertices, edges, or largest degrees and the efficiency of the strategies.

5.2 Bounds

To compare the bounds with each other, we assign each of them a score. We use two ways to score bounds: First, the ordered score, which assigns to the best bound the value one, and to the worst bound the value zero. The intermediate bounds get evenly spaced scores between zero and one. To be more precise, for n bounds $b_0, \ldots, b_{n-1}$ that take on values such that $b_i < b_{i+1}$ we define the ordered score s_o to be

$$s_o(b_i) = \frac{i}{n-1}$$

for lower bounds and

$$s_o(b_i) = \frac{n-1-i}{n-1}$$

for upper bounds. To better capture how close the individual bounds are to each other, we also calculate a proportional score s_p. Here, the best bound still gets a score of one, and worse bounds get a score in proportion to the best bound. For lower bounds b_i as above we define

$$s_p(b_i) = \frac{b_i}{b_{n-1}}.$$

For upper bounds we take the reciprocal so that every score is again between zero and one:

$$s_p(b_i) = \frac{b_0}{b_i}$$

Table 5.2 shows the score of all lower bounds averaged over all subproblems as well as all explored nodes within the respective search trees. We also list the average time to compute each bound. The largest degree is used in the branching step and branching on short paths is turned off. Because we have many subproblems that are small, we also determine the averages among larger instances that have more than 50 vertices.

Table 5.2: Comparison of lower bounds

Bound	all instances			large instances		
	s_o	s_p	time[μs]	s_o	s_p	time[μs]
a_l	0.057	0.869	0.25	0.161	0.618	0.81
$\tilde{s}$	0.105	0.793	0.25	0.067	0.407	0.81
$\tilde{s}^U$	0.352	0.930	0.25	0.578	0.822	0.81
$\bar{s}$	0.343	0.857	0.43	0.278	0.576	1.56
$\bar{s}^U$	0.583	0.980	0.34	0.732	0.935	1.33
$\bar{s}^U_C$	0.708	0.988	25.25	0.857	0.961	95.22
r	0.616	0.869	2187.49	0.356	0.619	7551.81
r_C	0.741	0.870	1949.48	0.481	0.621	7441.51
s	0.969	0.990	53658.46	0.977	0.997	58013.41

Table 5.3: Comparison of upper bounds

Bound	all instances			large instances		
	s_o	s_p	time[μs]	s_o	s_p	time[μs]
a_u	0.345	0.807	0.45	0.229	0.782	0.69
g_d	0.612	0.984	3513.74	0.635	0.989	4028.03
g_o	0.320	0.981	56180.73	0.343	0.984	65538.38
g_p	0.774	0.989	56307.50	0.776	0.990	62850.44

Among all instances, a_l is on average the weakest lower bound, but it can nevertheless come close to the better bounds if the instance is small. The most simple approximation to the subdivided star cover number $\tilde{s}$ is better in most cases, but falls behind a_l in larger graphs because it does not take the number of vertices that are already active into account. For smaller graphs, taking only unobserved vertices into account ($\tilde{s}^U$) and counting the degrees ($\bar{s}$) make for a similar improvement. However, for larger graphs the restriction to unobserved vertices is considerably more effective. Further improvents such as $\bar{s}^U$ and $\bar{s}^U_C$ make for better lower bounds, however the latter is the first one to take more than just a few microseconds to compute. Taking the relaxed integer linear program as a lower bound does not make for a good lower bound considering the much larger computation time. We observe that the time it takes to compute the subproblems is made up for by the faster computation for the individual linear programs, so r_C is not only a better bound than r, but is also faster to compute. As expected, we find that the subdivided star cover number is the best lower bound in the vast majority of cases, especially for larger graphs. This comes at a significant cost - for the largest subproblem, solving the integer linear program takes almost half a second.

As for the lower bounds, a_u is consistently the worst upper bound (Table 5.3), but it is also the only fast one. The three other bounds score similar values on average, however the bound based on greedily adding vertices by largest observation neighborhood g_o ranks lower more often. Picking vertices by proximity is slightly more effective than selecting the largest degree, however g_d is significantly faster to compute than g_o and g_p.

Because the subdivided star number s is our best lower bound, but is hard to compute, we want to compare it to its best lower bound $\bar{s}^U_C$. In figure 5.1a and 5.1b we plot the time to compute the respective bounds. We can see that both grow roughly linearly with the number of vertices. However, calculating s is about five hundred times slower than computing its lower bound. In figure 5.1c we show how close the lower bound comes to the actual value of s. For small graphs, $\bar{s}_C^U$ often is close to s. With a growing number of vertices the gap widens, although more data would be useful to see how strong the correllation actually is. Within our data set, the average degree and ratio of propagating vertices seem to not have a strong effect on neither the computation time nor the ratio of the two bounds.

To evaluate the runtime of the overall algorithm, we want to choose a selection of bounds that offer a good balance between closeness to γ_P and computational cost. The average runtimes fall in four groups:

very fast $a_l, \tilde{s}, \tilde{s}^U, \bar{s}, \bar{s}^U, a_u$

fast $\bar{s}_C^U$

medium r, r_C, g_d

slow s, g_o, g_p

Because the time needed to compute the bounds in a faster category is negligable compared to the next slower category, we add to each group all bounds from faster groups. We additionally reduce the number of bounds per group by removing those that are dominated by another bound in the same group. We get the following groups:

very fast $a_l, \bar{s}^U, a_u$

fast $a_l, \bar{s}_C^U, a_u$

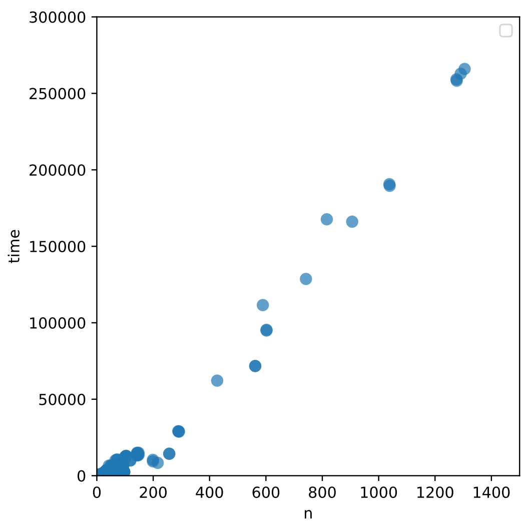
medium $a_l, \bar{s}_C^U, r_C, g_d$

slow s, r_C, g_d, g_o, g_p

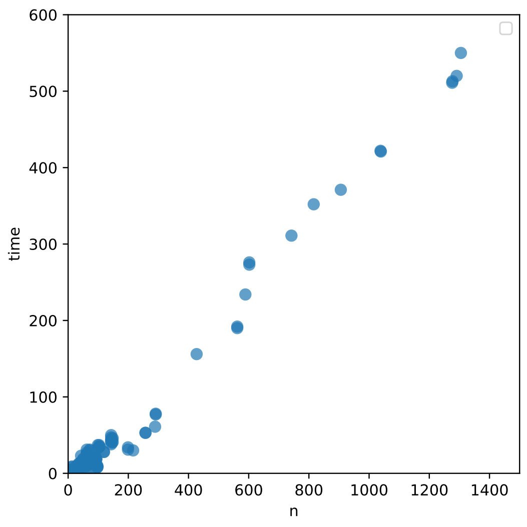
5.3 Performance of the Branching Algorithm

We compare the performance of our algorithm with the integer linear program presented by Bläsius and Göttlicher [BG23] which we solve using the gurobi solver for a selection of instances labelled A-F[1]. For each graph we compute a minimal power dominating set with each group of bounds aswell as the gurobi solver for reference. Again, the vertices for branching are picked in order of largest degree and branching on short paths is turned off. If the computation time exceeds two hours, we interrupt the algorithm. The results are shown in table 5.4. We list the number of vertices of the respective graphs, the largest degree and the number of non-propagating vertices (NP). Besides the total runtime, we also list the time spent in vertex selection, applying reduction rules and calculating bounds. The last two columns show the number of nodes the branching algorithm explores, and how many it discards.

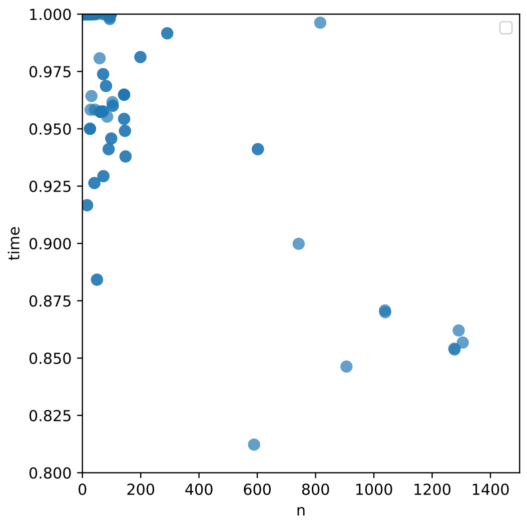
[1]The graphs A-F are subproblems of the following instances:
A: IEEE-300
B: case2383wp
C: IEEE-118
D: case2746-wop
E: psd-Western
F: case2869pegase



(a) Computation time for s in μs



(b) Computation time for $\bar{s}_C^U$ in μs



(c) Ratio of proportional scores between $\bar{s}_C^U$ and s

Figure 5.1: Comparison between s and $\bar{s}_C^U$

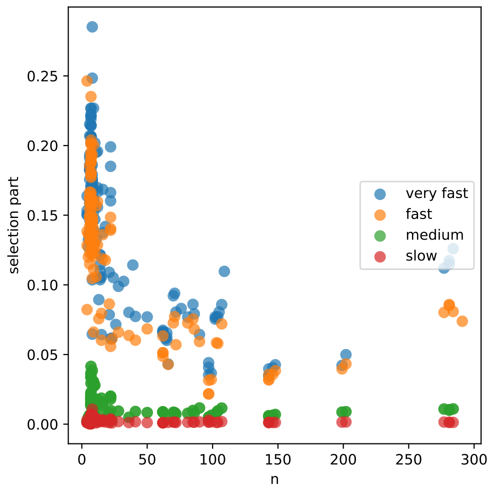
Small instances of fewer than 50 vertices can be solved quickly by our algorithm. For graphs like A and B, picking the vertex of largest degree is good enough that we arrive at the solution almost immediately. Here, time is spent mostly in reducing the graph after picking a vertex and verifying that no better solution exists. Computing medium or slow bounds is not necessary in these cases. Whether fast or only very fast bounds are computed does not matter, as the time needed is negligable compared to applying reduction rules. In these cases we beat the reference solution. For graph C, fast and very fast bounds are still the most useful, however they are no longer faster than gurobi. Using slow bounds, the algorithm is significantly slower but needs to explore about one quarter of the search space compared to faster bounds.

On larger graphs such as E, our algorithm still finds a solution but is much slower. Now choosing the group of fast bounds is clearly the best option. In comparison, very fast bounds are too far of the actual solution, so about five times more nodes need to be explored before the algorithm terminates. Medium and slow bounds on the other hand are too expensive to compute.

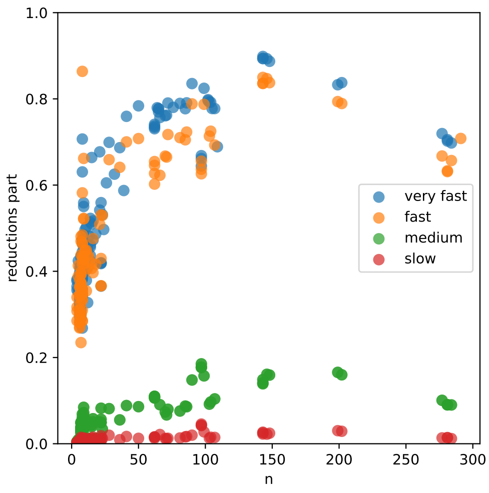
Despite being similarly large and having more non-propagating vertices, graph E takes about ten times longer to solve than graph D. This indicates that the precise structure of a graph has a strong impact on its difficulty that the graph parameters do not capture on their own. Slow bounds now take too much time to complete in the alloted time. This extends to all groups of bounds for graph F, which is the largest among the listed instances.

Which sections of the algorithm takes most time depending on group of bounds is displayed in figure 5.2. Selecting the next vertex takes fairly little time regardless of the selected group of bounds. Very fast and fast bounds lead to most time being spent in reduction rules whereas medium and slow bounds lead to more time used for bounds computation. The exception are small graphs, where time is spent more evenly in the three categories although the data varies strongly between instances.

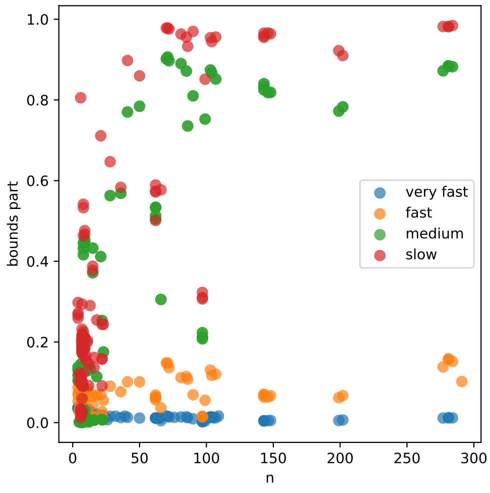
In figure 5.3 we plot the total time our algorithm uses to compute γ_P using fast bounds on a log-linear scale. We see that the computation time grows exponentially with the number of vertices in the graphs, with computation time growing about ten times for fifty additional vertices. The correlation in 5.3b is less strong, but a larger average degree seems to indicate that the problem is more difficult to solve. The portion of vertices that are non-propagating does not have a clear effect on the runtime.



(a) vertex selection

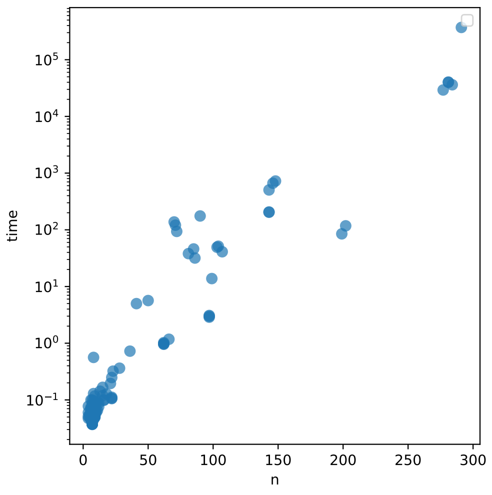


(b) reduction rules

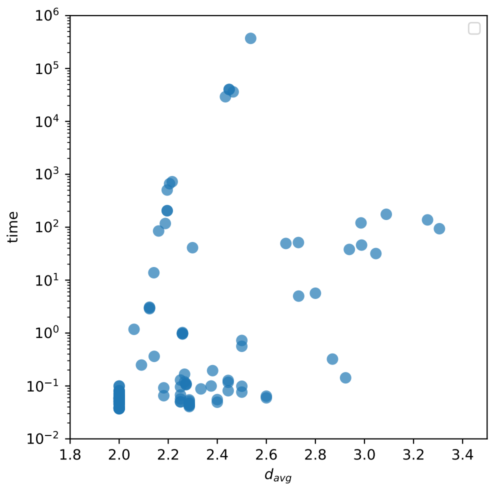


(c) bounds

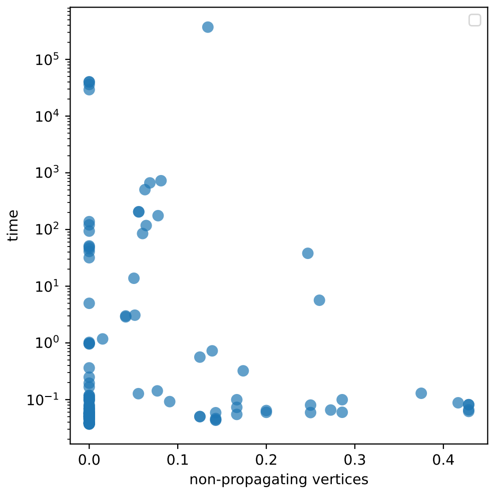
Figure 5.2: Proportion of computation time that is spent in different sections for each group of bounds



(a) number of vertices



(b) average degree



(c) part of vertices that are non-propagating

Figure 5.3: Computation time in milliseconds different graph properties

Table 5.4: Runtime of the Algorithm using different groups of bounds

Graph	Parameters			Solver	Time[ms]				#Nodes	
	n	$d_{\max}$	NP		total	select	reduct	bound	expl.	disc.
A	5	3	0	v. fast	0.06	< 0.01	0.02	< 0.01	1	2
				fast	0.05	< 0.01	0.02	< 0.01	1	2
				medium	3.33	< 0.01	0.02	< 0.01	1	2
				slow	5.36	< 0.01	0.02	0.5	1	2
				gurobi	2.0					
B	22	3	0	v. fast	0.23	0.01	0.13	< 0.01	2	1
				fast	0.25	0.01	0.13	< 0.01	2	1
				medium	1.54	0.02	0.13	0.39	1	2
				slow	7.31	0.02	0.13	1.78	1	2
				gurobi	3.0					
C	85	9	0	v. fast	45	4	35	< 1	174	173
				fast	46	3	32	5	163	162
				medium	364	4	31	317	153	154
				slow	944	1	15	902	44	43
				gurobi	7					
D	284	6	0	v. fast	132262	16655	92276	1542	189856	169970
				fast	36154	2923	23751	5478	33872	29757
				medium	232194	2540	20896	204882	25651	25652
				slow	366307	469	4378	360572	4400	4351
				gurobi	39					
E	291	6	39	v. fast	1994350	159476	1114630	10954	1476810	1216341
				fast	370828	27387	262631	37770	283251	283251
				medium	3515560	37839	276191	3144420	285409	282704
				slow	timeout				125876	16639
				gurobi	80					
F	589	10	106	v. fast	timeout				5020450	0
				fast	timeout				2414978	0
				medium	timeout				246256	0
				slow	timeout				30634	0
				gurobi	439					

6 Conclusion

In this thesis, we presented an alternative algorithm for solving power dominating set using a branch and bound approach. For this algorithm, we derived multiple lower and upper bounds for the power domination number γ_P. We introduced the concept of the subdivided star cover of a graph and proved its relation to power dominating set and the observation graph in particular. With the subdivided star cover number s we provide a lower bound for γ_P that is in itself NP-hard to compute and give an integer linear program to compute it. Additionally we discussed two branching strategies, one using the inclusion or exclusion of a vertex in the power dominating set. Alternatively we explored branching on short paths, a wide branching strategy that can be used to discard some parts of the search space early. Lastly, we evaluated the bounds used in our algorithm as well as its overall performance on graphs common in related literature.

For small inputs our algorithm is faster than solving the integer linear program given by Bläsius and Göttlicher [BG23]. In these cases our strategies for picking vertices can select a minimal power dominating set with only a few nodes being explored. The minimality of the selected set is then proven by lower bounds that are both fast to compute and a good approximation for γ_P on small instances. However for larger instances, our branch and bound approach is slower than the reference solution by multiple orders of magnitude. Next we want to discuss a few options for improving the performance of our algorithm.

Some of the lower bounds we presented could be further sped up by using a data structure for the graph that stores certain additional data like the largest degree. This could bring the asymptotic runtime for computing $\tilde{s}$ from linear time to constant time. A similar, but more elaborate approach could bring the same improvement to $\tilde{s}^U$. However, since these bounds can already be computed very fast, the improvement would like only affect the performance of our algorithm on instances where it is already efficient. In section 5.2 we grouped our bounds by similar performance. Further evaluation could show that we can get similar results while discarding some bounds. More generally, the algorithm could compute the bounds iteratively and discard the instance as soon as the first bound exceeds the allowed threshold. This is in contrast to our current approach, which computes the minimum or maximum of all the bounds, and then discards the instance if possible. But this again will not meaningfully improve performance on large graphs, since even our best bounds are not close enough to γ_P. In addition, it could be interesting to evaluate the bound given by relaxing the integer linear program for the subdivided star cover number. Currently, computing s is only about one order of magnitude faster than computing γ_P using the reference solution. This could be a worthwile compromise between the quality of the bound and its computation time.

The most important direction of research that could make branch and bound algorithms a competing approach is therefore the search for tighter bounds on the power domination number. Here, study of cases in which the gap between our bounds and the minimal solution is especially wide could be useful. However the question remains whether bounds that are good enough for efficient branching are also necessarily hard to compute. We showed that even computing the subdivided star cover number s is NP-hard, and still is not close enough to γ_P to make our algorithm feasible on larger instances.

In general, power dominating set differs from two related problems that make it harder to derive bounds. Comparing it to dominating set, it lacks useful monotonicity properties (see section 3.1) and in comparison to hitting set, the amount of vertices that can be observed by one active vertex is not constrained by its degree as we explain in section 3.2. A wholly different approach to a branching algorithm for power dominating set could be to branch on the direction of the edges in the observation graph instead of the inclusion of vertices. However we believe that this approach would be less useful for two reasons: First, the search space grows form 2^n nodes to 3^m because each edge can be present in two directions or absent. Furthermore, both representations of the graph can be translated into each other as long as a vertex order is given, giving rise to the same bounds.

In conclusion, our algorithm is not competitive with different approaches on larger graphs but can outperform the reference solution on small instances. However, we gained theoretical insights into bounds for power dominating set as well as showing its relation to other problems.

Bibliography

[BBE18]	C. Bozeman, B. Brimkov, and C. et al. Erickson. "Restricted power domination and zero forcing problems". In: *Journal of Combinatorial Optimization* (2018). DOI: *https://doi.org/10.1007/s10878-018-0330-6*.
[BFSW]	Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. "An Efficient Branch-and-Bound Solver for Hitting Set". In: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 209–220. eprint: *https://epubs.siam.org/doi/pdf/10.1137/1.9781611977042.17*.
[BG23]	Thomas Bläsius and Max Göttlicher. "An Efficient Algorithm for Power Dominating Set". 2023.
[BH05]	Dennis Brueni and Lenwood Heath. "The PMU placement problem". In: *SIAM J. Discrete Math.* Volume 19 (Jan. 2005), pp. 744–761. DOI: *10.1137/S0895480103432556*.
[Bin11]	H. Binkele-Raible D.; Fernau. "An Exact Exponential Time Algorithm for Power Dominating Set". In: (2011). DOI: *https://doi.org/10.1007/s00453-011-9533-2*.
[BJ18]	Csilla Bujtás and Marko Jakovac. "Relating the total domination number and the annihilation number of cactus graphs and block graphs". In: *Ars Mathematica Contemporanea* Volume 16 (Nov. 2018), pp. 183–202. DOI: *10.26493/1855-3974.1378.11d*.
[CTF00]	A. Caprara, P. Toth, and M Fischetti. "Algorithms for the Set Covering Problem". In: *Annals of Operations Research* (2000). DOI: *https://doi.org/10.1023/A:1019225027893*.
[Dor20]	Paul Dorbec. "Power domination in graphs". In: *Topics in Domination in Graphs*. 2020. DOI: *10.1007/978-3-030-51117-3_16*.
[DVV16]	Paul Dorbec, Seethu Varghese, and Ambat Vijayakumar. *Heredity for generalized power domination*. 2016. arXiv: *1603.07243*.
[Eve+03]	G. Even, N. Garg, J. Könemann, R. Ravi, and A. Sinha. "Covering Graphs Using Trees and Stars". In: *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*. Edited by Sanjeev Arora, Klaus Jansen, José D. P. Rolim, and Amit Sahai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 24–35. ISBN: 978-3-540-45198-3.
[HHHH02]	Teresa W. Haynes, Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Michael A. Henning. "Domination in Graphs Applied to Electric Power Networks". In: *SIAM Journal on Discrete Mathematics* Volume 15 (2002), pp. 519–529. eprint: *https://doi.org/10.1137/S0895480100375831*.
[JV20]	Raka Jovanovic and Stefan Voss. "The fixed set search applied to the power dominating set problem". In: *Expert Systems* Volume 37 (2020), e12559. eprint: *https://onlinelibrary.wiley.com/doi/pdf/10.1111/exsy.12559*.

[Kar72] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.* Edited by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: *10.1007/978-1-4684-2001-2_9*.

[KU16] Kolja Knauer and Torsten Ueckerdt. “Three ways to cover a graph”. In: *Discrete Mathematics* Volume 339 (2016), pp. 745–758. ISSN: 0012-365X. DOI: *https://doi.org/10.1016/j.disc.2015.10.023*.

[MJSS16] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. “Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning”. In: *Discrete Optimization* Volume 19 (2016), pp. 79–102. ISSN: 1572-5286. DOI: *https://doi.org/10.1016/j.disopt.2016.01.005*.

[Moh13] Ahmad Nasir Mohamad. “THE COMBINATION OF SPIDER GRAPHS WITH STAR GRAPHS FORMS GRACEFUL”. In: *International Journal of Advanced Research in Engineering and Applied Sciences* (2013).

[PP16] A. Panpa and T. Poomsa-ard. “On Graceful Spider Graphs with at Most Four Legs of Lengths Greater than One”. In: *Journal of Applied Mathematics* (2016).

[Şah22] Bünyamin Şahin. “New network entropy: The domination entropy of graphs”. In: *Information Processing Letters* Volume 174 (2022), p. 106195. ISSN: 0020-0190. DOI: *https://doi.org/10.1016/j.ipl.2021.106195*.

[ZKC06] Min Zhao, Liying Kang, and Gerard J. Chang. “Power domination in graphs”. In: *Discrete Mathematics* Volume 306 (2006), pp. 1812–1816. ISSN: 0012-365X. DOI: *https://doi.org/10.1016/j.disc.2006.03.037*.