Computational Methods for Domination Problems

by

William Herbert Bird

B.Sc., University of Victoria, 2011

M.Sc., University of Victoria, 2013

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Computational Methods for Domination Problems

by

William Herbert Bird

B.Sc., University of Victoria, 2011

M.Sc., University of Victoria, 2013

Supervisory Committee

Dr. Wendy Myrvold, Supervisor

(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member

(Department of Computer Science)

Dr. Kieka Mynhardt, Outside Member

(Department of Mathematics and Statistics)

Supervisory Committee

Dr. Wendy Myrvold, Supervisor

(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member

(Department of Computer Science)

Dr. Kieka Mynhardt, Outside Member

(Department of Mathematics and Statistics)

ABSTRACT

For a graph G, the *minimum dominating set* problem is to find a minimum size set S of vertices of G such that every vertex is either in S or adjacent to a vertex in the set. The decision version of this problem, which asks whether G has a dominating set of a particular size k, is known to be NP-complete, and no polynomial time algorithm to solve the problem is currently known to exist. The *queen domination problem* is to find the minimum number of queens which, collectively, can attack every square on an $n \times n$ chess board. The related *border queen problem* is to find such a collection of queens with the added restriction that all queens lie on the outer border of the board. This thesis studies practical exponential time algorithms for solving domination problems, and presents an experimental comparison of several different algorithms, with the goal of producing a broadly effective

general domination solver for use by future researchers. The developed algorithms are then used to solve several open problems, including cases of the queen domination problem and the border queen problem. In addition, new theoretical upper bounds are presented for the border queen problem for some families of queen graphs.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables viii

List of Figures xi

List of Algorithms xiv

1 Introduction 1

1.1 Definitions 2

1.1.1 Graphs 3

1.1.2 Dominating Sets 3

1.1.3 Independent Dominating Sets 4

1.2 Complexity and Parameterized Complexity 5

1.3 Related Computational Problems 9

1.4 Algorithms to Compute Minimum Dominating Sets 11

2 Queen Graphs and Other Interesting Graph Classes 15

2.1 The Queen Domination Problem 16

2.2 Irredundant Sets 18

2.3 The Border Queen Problem 20
2.4 Kneser Graphs 26
2.5 Covering Codes and Football Pools 26
2.6 Triangle Grid Graphs 30
2.7 Hex Rook Graphs 31
2.8 Cartesian Products of Cycles 32

3 Algorithms **34**
3.1 Backtracking Framework 35
3.1.1 Bounding Condition 37
3.1.2 Vertex Selection 37
3.1.3 Neighbour Ordering 37
3.2 Bounding With Fixed Vertex Ordering 38
3.2.1 Implementation: Algorithm 3.2 38
3.3 Domination Degree Algorithms 41
3.3.1 Domination Degree Multiset 44
3.3.2 Candidate Degree Priority Queue 47
3.3.3 Implementation: Algorithm 3.5 58
3.4 Max Dominator Degree Algorithms 61
3.4.1 MDD Ranking Data Structure 63
3.4.2 Implementation: Algorithm 3.7 66

4 Experimental Evaluation of Domination Algorithms **70**
4.1 Input Graph Dataset 72
4.2 Methodology 73
4.2.1 Mitigating the impact of 'luck' 75
4.3 Fixed-Ordering Implementations 77

4.4 Domination Degree Implementations 81
4.4.1 Single Aspect Comparisons 83
4.5 Max Dominator Degree Implementations 88
4.5.1 Single Aspect Comparison 88
4.6 Comparison of Framework Algorithms 96
4.7 Comparison with SageMath 100
4.8 Choosing Representative Algorithms 103
4.8.1 Overall Variant Comparison 105
4.8.2 Comparison of Variants by Graph Family 107

5 New Domination Results for Queen Problems 113
5.1 Computing Independent Dominating Sets 114
5.2 Splitting Computation Among Processes 115
5.3 Counting Solutions up to Isomorphism 118
5.4 Certificates of Independent Dominating Sets 119
5.5 Rotated Border Constructions 122
5.5.1 Searching for Minimum RBCs 139
5.5.2 Summary of Border Queen Results 142

6 `Unidom` 149
6.1 The `unidom` Architecture 149
6.2 Input Source 151
6.3 Preprocessing Filters 153
6.4 Solver 156
6.5 Output Proxy 158

7 Conclusions and Future Research 161

Bibliography **165**

List of Tables

Table 2.1 Domination Numbers of Kneser Graphs 27
Table 2.2 Solutions to the Football Pool Problem for $n = 1, \ldots, 10$ 29
Table 2.3 Domination Numbers of Hypercubes Q_n for $n = 1, \ldots, 11$ 29
Table 2.4 Domination Numbers of Hex Rook Graphs 32

Table 4.1 Optimization Experiment Input Graphs 73
Table 4.2 Fixed Ordering Running Times: Covering Code Graphs 79
Table 4.3 Fixed Ordering Running Times: Hex Rook Graphs 79
Table 4.4 Fixed Ordering Running Times: Kneser Graphs 79
Table 4.5 Fixed Ordering Running Times: Knight Graphs 80
Table 4.6 Fixed Ordering Running Times: Cartesian Products of Cycles 80
Table 4.7 Fixed Ordering Running Times: Queen Graphs 80
Table 4.8 Fixed Ordering Running Times: Triangle Grid Graphs 81
Table 4.9 DD Bounding: Summary of maximum time ratios for all aspects on all graph families 85
Table 4.10 DD Bounding: Summary of maximum total call ratios for all aspects on all graph families 88
Table 4.11 MDD Bounding: Summary of maximum time ratios for all aspects on all graph families 95
Table 4.12 MDD Bounding: Summary of maximum total call ratios for all aspects on all graph families 96

Table 4.13 Comparison of Framework Algorithms - Maximum Times: Covering Code Graphs 97
Table 4.14 Comparison of Framework Algorithms - Maximum Times: Hex Rook Graphs 98
Table 4.15 Comparison of Framework Algorithms - Maximum Times: Kneser Graphs 98
Table 4.16 Comparison of Framework Algorithms - Maximum Times: Knight Graphs 99
Table 4.17 Comparison of Framework Algorithms - Maximum Times: Cartesian Products of Cycles 99
Table 4.18 Comparison of Framework Algorithms - Maximum Times: Queen Graphs 99
Table 4.19 Comparison of Framework Algorithms - Maximum Times: Triangular Grid Graphs 100
Table 4.20 SageMath vs. Framework 3.1 - Maximum Times: Covering Code Graphs101
Table 4.21 SageMath vs. Framework 3.1 - Maximum Times: Hex Rook Graphs . 101
Table 4.22 SageMath vs. Framework 3.1 - Maximum Times: Kneser Graphs . . . 101
Table 4.23 SageMath vs. Framework 3.1 - Maximum Times: Knight Graphs . . . 102
Table 4.24 SageMath vs. Framework 3.1 - Maximum Times: Cartesian Products of Cycles 102
Table 4.25 SageMath vs. Framework 3.1 - Maximum Times: Queen Graphs . . . 102
Table 4.26 SageMath vs. Framework 3.1 - Maximum Times: Triangular Grid Graphs102
Table 4.27 Best 10 average maximum time fractions of tested algorithms on the entire input dataset. 106
Table 4.28 Best 10 average maximum time fractions of tested algorithms on the 20 moderately difficult graphs in the input dataset. 106
Table 4.29 Maximum times of Framework 3.1 variants on Covering Code graphs. 108
Table 4.30 Maximum times of Framework 3.1 variants on Hex Rook graphs. . . . 109
Table 4.31 Maximum times of Framework 3.1 variants on Kneser graphs. 110

Table 4.32 Maximum times of Framework 3.1 variants on Knight graphs. 110
Table 4.33 Maximum times of Framework 3.1 variants on Cartesian Products of Cycles. 111
Table 4.34 Maximum times of Framework 3.1 variants on Queen graphs. 111
Table 4.35 Maximum times of Framework 3.1 variants on Triangular Grid graphs. 112

Table 5.1 Domination Numbers of Queen Graphs 114
Table 5.2 Number of Minimum Dominating Sets of Queen Graphs up to Isomorphism 118
Table 5.3 Number of Minimum Independent Dominating Sets of Queen Graphs up to Isomorphism 119
Table 5.4 Minimum Border Dominating Sets of Queen Graphs up to Isomorphism 119
Table 5.5 Summary of Border Domination Parameters 143

List of Figures

Figure 2.1 3×3 chess board and Queen graph of order 3 17

Figure 2.2 A maximal irredundant set of C_5 19

Figure 2.3 Examples of the construction used in the proof of Theorem 2.8. . . . 23

Figure 2.4 Counterexample of an assertion by Burchett in [9]. 24

Figure 2.5 Triangle grid and hex rook graphs of order 3 30

Figure 3.1 An 8×8 board with two queens. 42

Figure 3.2 A 10×10 board with three queens. 61

Figure 4.1 DD Bounding: Histogram of pairwise ratios for Min. CD vs. Max. CD vertex selection. 86

Figure 4.2 DD Bounding: Histogram of pairwise ratios for ascending vs. descending neighbour order. 86

Figure 4.3 DD Bounding: Histogram of pairwise ratios for force stop optimization disabled vs. enabled. 87

Figure 4.4 DD Bounding: Histogram of pairwise ratios for bound rechecking optimization disabled vs. enabled. 87

Figure 4.5 MDD Bounding: Histogram of pairwise ratios for Min. CD vs. Max. CD vertex selection. 90

Figure 4.6 MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Min. CD vertex selection. 90

Figure 4.7 MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Max. CD vertex selection. 91
Figure 4.8 MDD Bounding: Histogram of pairwise ratios for Max. MDD vs. Min. CD vertex selection. 91
Figure 4.9 MDD Bounding: Histogram of pairwise ratios for Max. MDD vs. Max. CD vertex selection. 92
Figure 4.10 MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Max. MDD vertex selection. 92
Figure 4.11 MDD Bounding: Histogram of pairwise ratios for ascending vs. descending neighbour order. 93
Figure 4.12 MDD Bounding: Histogram of pairwise ratios for force stop optimization disabled vs. enabled. 93
Figure 4.13 MDD Bounding: Histogram of pairwise ratios for bound rechecking optimization disabled vs. enabled. 94

Figure 5.1 Two strategies for splitting a computation among multiple processes (indicated by different colours). 117
Figure 5.2 An independent dominating set of Queen (19) with size 11. 120
Figure 5.3 An independent dominating set of Queen (20) with size 11. 120
Figure 5.4 An independent dominating set of Queen (22) with size 12. 121
Figure 5.5 An independent dominating set of Queen (23) with size 13. 121
Figure 5.6 An independent dominating set of Queen (24) with size 13. 122
Figure 5.7 Examples of two canonical RBCs on $n = 11$. 130
Figure 5.8 Examples of minimum RBCs for $n \in \{1, 2, 3, 4, 5, 6\}$. 145
Figure 5.9 A minimum RBC of size 12 of Queen (14). 146

Figure 6.1 Diagram of the 'pipeline' used by `unidom` computations. 150

Figure 6.2 Example of the adjacency list representation of TG (3). 152

List of Algorithms

3.1 Framework for a Backtracking Search 36
3.2 Backtracking Algorithm using Bounding Strategy 3.1 40
3.3 Operations of the Domination Degree Multiset Structure 48
3.4 Operations of the Candidate Degree Priority Queue Structure 54
3.5 Backtracking Algorithm using Bounding Strategy 3.3 59
3.6 MDD-based bound on the number of vertices needed to complete a dominating set. 63
3.7 Backtracking Algorithm using Bounding Strategy 3.6 68

Chapter 1

Introduction

Consider the problem of placing eight queens on a standard chessboard such that no two queens can attack each other. This problem, and its generalization, the *n-queens problem*, originated as a logic puzzle in 1848 [4]. Problems like the 8-queens problem, or the 15-puzzle, from the same time period [39], might have been idle amusements for mathematicians at the time, much like Sudoku puzzles are today. Indeed, the first comprehensive theoretical treatment of the n-queens problem appears to be in *Mathematical Recreations and Essays* by W. W. Rouse Ball, originally written in 1892 [55]. Many such puzzles went unsolved for years, and constructing a solution would have been a sign of high intellect or 'cleverness', since, in many cases, no solution was within reach of the creator of the puzzle. At the time, when clever humans were the only computing machines, these 'mathematical recreations' would have been as computationally difficult as today's open research problems.

The n-queens problem and the 15-puzzle are now given as exercises to thousands of first or second year computer science students every year. Displaying a solution is no longer a sign of cleverness; instead, the cleverness comes with crafting a computer algorithm to solve the problem conveniently. In fact, solving one of these 19th century puzzles by hand today might be considered a foolish waste of time when a computer could find a solution in a frac-

tion of a second. In many ways, the classical 'mathematical recreations' only truly became recreational when electronic computers superseded human ones. For today's 'hard' computational problems, most of the human cleverness involves designing algorithms which will solve (or approximately solve) the problem using reasonable resources (particularly memory and running time).

This thesis studies practical methods for solving one particular hard problem called the *dominating set* problem. Several new algorithms to solve the problem are given, and experimental data is presented which shows that the algorithms are practical on several classes of inputs for which existing methods are impractical. The new algorithms are used to solve several open cases of a modern descendant of the classic 8-queens problem called the *queen domination problem*, along with similar problems. A restricted variant of the queen domination problem, the *queen border domination problem*, is also studied, and various new results–both theoretical and computational–are given for that problem, including a conjectured characterization of the solution to the border queen problem for a large number of cases. Finally, the new algorithms are combined into a software tool for use by future researchers.

Section 1.1 establishes the mathematical background used for the rest of this document and defines the concept of a dominating set. Sections 1.2 - 1.4 define the computational domination problem and survey previous results on domination problems in computer science. The structure of the remaining chapters of this thesis is described at the end of Section 1.4.

1.1 Definitions

In general, the notation and definitions for graphs in this document are based on the conventions established by West [65]. The algorithms and data structures studied in this thesis

are intended for use with a random access machine with a word size sufficient to store all of the numerical values computed in each algorithm. Asymptotic bounds on time and space complexity are stated in terms of this (fixed) word size.

1.1.1 Graphs

A *graph* G is an ordered pair (V, E) containing a set V of *vertices* and a collection E of *edges* which correspond to unordered pairs uv of vertices in V. The vertex and edge sets of a particular graph can be denoted by the functional notation $V(G)$ and $E(G)$. Given a vertex v, a vertex u such that an edge uv exists is called a *neighbour* of v. The two endpoints u and v of an edge uv are said to be *adjacent*. When the pairs in E are ordered, G is called a *directed graph*, and the pairs in E are usually called *arcs*. Otherwise, the graph G is *undirected*. If E is a multiset (which may contain a pair uv more than once), G is called a *multigraph*. A *loop* in a graph is an edge of the form vv from a vertex v to itself. A graph with no loops and no duplicate edges is called *simple*, and in this document, all graphs will be simple and undirected. The *degree* of a vertex $v \in V(G)$ is equal to the number of edges incident to v, and the maximum degree over all vertices in a graph G is denoted by $\Delta(G)$ (or just Δ in cases where the graph is clear from context).

1.1.2 Dominating Sets

Let G be a graph on $|V(G)| = n$ vertices. The *closed neighbourhood* of a vertex $v \in V(G)$, denoted by $N[v]$, is a set containing v and all neighbours of v. If $S \subseteq V(G)$ is a set of vertices, then the *closed neighbourhood* of S is defined to be

$$N[S] = \bigcup_{v \in S} N[v].$$

A *dominating set* of a graph G is a set of vertices $D \subseteq V(G)$ such that every vertex $v \in V(G)$ is either in D or adjacent to a vertex in D. Equivalently, a set $D \subseteq V(G)$ is a dominating set if and only if $N[D] = V(G)$. The minimum number of vertices in a dominating set of G is called the *domination number* of G and is denoted by $\gamma(G)$.

The concept of domination can also be defined for individual vertices v or arbitrary subsets $S \subseteq V(G)$. A vertex v dominates all of the vertices in $N[v]$ and a set $S \subseteq V(G)$ dominates all of the vertices in $N[S]$.

1.1.3 Independent Dominating Sets

An *independent set* of a graph G is a subset $S \subseteq V(G)$ such that no two vertices in S are neighbours. Any maximal independent set S of a graph G must also be a dominating set of G, since if any vertex $v \in V(G)$ is not dominated by S, the set $S \cup \{v\}$ would be an independent set (which contradicts the maximality of S). This property, combined with the fact that every graph G on at least one vertex must have an independent set, implies that every non-trivial graph must have an independent set which is also a dominating set. The minimum size of an independent dominating set is called the *independent domination number* of G and is denoted by $i(G)$. The quantity $i(G)$ is not to be confused with the overall maximum size of an independent set (the *independence number* of G), which is typically denoted by either $\alpha(G)$ or $\beta(G)$. There is considerable disagreement over which symbol to use for the independence number of a graph. Many of the articles referenced in this thesis use $\beta(G)$ or $\beta_0(G)$ instead of $\alpha(G)$. In this thesis, the notation $\alpha(G)$ is used consistently, which may result in some notational inconsistencies with the referenced material.

1.2 Complexity and Parameterized Complexity

A *decision problem* is a computational problem which asks, for a given input string x, whether or not x is a member of some language L. Decision problems are often phrased in less abstract terms as problems for which there is a 'yes' or 'no' answer. An algorithm A with an input x is said to *accept* a language L if A outputs 'yes' for all $x \in L$ and 'no' for all $x \notin L$.

The complexity class NP contains all languages which are accepted by a non-deterministic algorithm in polynomial time, or, equivalently, all languages L for which there exists some deterministic polynomial time algorithm such that

$$L = \{x \in \Sigma^* : A \text{ accepts } (x, c) \text{ for some } c \in \Sigma^*\}$$

where Σ is an alphabet [36]. The string c is normally called a *certificate* or *witness*.

Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be languages over the alphabets Σ_1^* and Σ_2^*, respectively. L_1 is said to be *reducible* to L_2 if there exists a function $f : \Sigma_1^* \to \Sigma_2^*$ such that, for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$. Furthermore, if the mapping f can be computed by some polynomial time algorithm A, then L_1 is said to be *polynomial time reducible* to L_2, denoted by $L_1 \leq_p L_2$. It should be noted that there are various types of polynomial time reductions; the one defined here is often called a 'Karp-reduction' [36, 21]. The notion of reducibility can be applied to two problems P_1 and P_2 by the applying the definition above to the languages corresponding to instances of each problem encoded in some alphabet (for example, binary) [36].

A decision problem P (which may not itself lie in NP) is said to be *NP-hard* if, for all $P' \in$ NP, P is polynomial time reducible to P' [3]. Note that the notion of NP-hardness can also be defined for non-decision problems (such as counting or optimization problems), but the decision-based version is sufficient for this thesis. If $P \in$ NP and P is NP-hard, then P

is said to be *NP-complete*. To prove that a problem P is NP-hard, it is sufficient to prove that some NP-hard problem can be reduced to P.

The classical Dominating Set decision problem takes a pair (G, k) consisting of a graph G on n vertices and an integer k and asks whether G contains a dominating set of size at most k. Dominating Set is in NP, since a dominating set S can be verified in polynomial time by checking that $S \subseteq V(G)$ and that for each $v \in V(G)$, $N[v] \cap S \neq \emptyset$. The Dominating Set problem is also NP-complete, even under comparatively tight restrictions on the input, such as restriction to planar graphs with maximum degree three [29]. Deciding whether a graph G has an independent dominating set of a given size k is also NP-complete.

In addition to examining the complexity of the general form of a computational problem P, the problem can be subdivided into classes by a *parameterization*, which is a positive integer-valued function $\mathrm{Par}(x)$ such that, for each instance x of P, $\mathrm{Par}(x)$ is computable in polynomial time [1]. With respect to a particular parameterization $\mathrm{Par}(x)$, the problem P is said to be *fixed-parameter tractable* if there exists an algorithm A which can give a 'yes' answer to an instance x of P in time asymptotically bounded by

$$f(\mathrm{Par}(x))|x|^c$$

where f is any integer-valued computable function and c is a constant [36]. Note that when the parameter $\mathrm{Par}(x)$ is held fixed, the algorithm A asks membership in P in polynomial time.

Domination problems have a particular theoretical significance in the study of parameterized complexity, where the Dominating Set problem, parameterized by the set size k, is a complete problem for the complexity class $W[2]$, which is part of the W-hierarchy of parameterized complexity classes [53, 21]. Many of the other classical graph theoretical problems,

[1] For technical reasons which are not relevant to this work, Hromkovič [36] places other constraints on parameterizations to exclude degenerate cases.

such as INDEPENDENT SET and CLIQUE, along with the venerable 3-SAT boolean satisfiability problem, are complete for level $W[1]$. The levels of the W-hierarchy are defined by a class of problems called WEIGHTED WEFT t DEPTH h CIRCUIT SATISFIABILITY, denoted by WCS(t, h). For each level $W[t]$ of the W-hierarchy, WCS(t, h) is a complete problem for $W[t]$.

The problems in WCS(t, h) are parameterized variants of circuit satisfiability with constraints on the depth and 'weft' of circuits (from which the W-hierarchy gets its name). The depth of a circuit is the maximum number of gates between an input pin and an output pin. In the model used to define the W-hierarchy, logic gates in the circuit are permitted to have any number of input pins (so, for example, a disjunction of k inputs can be modelled by a single OR gate). Gates whose fan-in is bounded above by some pre-determined constant c (which is not dependent on the size or properties of a particular input) are said to be 'small', and gates whose fan-in is potentially unbounded (for example, because the fan-in is a function of the input size) are said to be 'large'. The *weft* of a circuit is the maximum number of 'large' gates between an input pin and an output pin of the circuit [21].

An instance of WCS(t, h) consists of a pair (C, k) where C is an encoding of a circuit and k is an integer, and $(C, k) \in$ WCS(t, h) if C has weft t, depth h and there exists a satisfying assignment of C with at most k inputs set to one. The latter constraint is the result of parameterizing the satisfiability problem by the Hamming weight k of the solution, and is significant for studying parameterized complexity.

Membership in the W-hierarchy depends on the notion of a parameterized reduction, which is a polynomial-time reduction between two parameterized languages L and L' where where an instance (P, k) (parameterized by k) can be reduced in polynomial time to an instance (P', k') (parameterized by k') such that $(P, k) \in L$ if and only if $(P', k') \in L'$, with the additional constraint that the parameter k' must be determined strictly by the value of k, rather than by any of the contents of the problem instance P [21]. The added constraint on

k and k' is what differentiates the reduction from the standard polynomial-time reductions used to prove NP-hardness.

A parameterized problem P can be shown to be a member of the class $W[t]$ by giving a parameterized reduction from P to WCS(t, h) for some (fixed) h. The venerable 3-SAT problem decides whether a boolean formula in 3-CNF form (consisting of a conjunction of clauses containing exactly 3 literals) has a satisfying assignment. A parameterization of 3-SAT by the Hamming weight k of the satisfying assignment can be shown to be in $W[1]$ by constructing a circuit with 3-input OR gates to compute each clause and an unbounded-input AND gate to take the conjunction of all clauses. This circuit has weft one and depth two, and any solution to the 3-SAT instance with k 1-bits will correspond to a solution of WCS$(1, 2)$ with k 1-bits. Additionally, the circuit can be constructed in polynomial time, so the requirements of the parameterized reduction are met. The Independent Set problem has also been shown to lie in $W[1]$ [21].

The set FPT of all problems which are fixed-parameter tractable is equivalent to $W[0]$ [21]. A set $S \subseteq V(G)$ is a *vertex cover* of a graph G if every edge of G has at least one endpoint in S. The computational Vertex Cover problem takes a graph G and an integer k and asks whether there exists a vertex cover of G with size at most k. Vertex Cover, parameterized by the size k of the vertex cover, is fixed parameter tractable: Whether or not a graph G on n vertices has a vertex cover of size at most k can be decided in $O(2^k n)$ time. As mentioned previously, Independent Set, parameterized by the set size, is complete for $W[1]$ and Dominating Set, parameterized by set size, is complete for $W[2]$ [21]. The problem of finding an independent dominating set is also known to be $W[2]$-hard, due to a result by Downey, Fellows, McCartin and Rosamund [22] which gave a parameterized reduction of the dominating set problem to the independent dominating set problem[2].

Although Independent Set is complete for $W[1]$, the restriction of Independent

[2]Note that the reduction in question was given as part of a larger proof of a more significant result regarding approximability of dominating sets.

Set to planar graphs is fixed-parameter tractable [21]. Similarly, although Dominating Set is complete for $W[2]$, the restriction of Dominating Set to planar graphs is fixed-parameter tractable [2, 28]. The first complete proof of an FPT algorithm for computing dominating sets of planar graphs was given by Alber et al. [2] and had running time $O(8^k n)$ to compute a dominating set of size k on a planar graph with n vertices. This result was extended to produce a fixed-parameter tractable algorithm for finding dominating sets of graphs embeddable in surfaces of any (fixed) genus by Ellis, Fan and Fellows [23]. A different analysis of the algorithm in [2] later revealed that the running time was $O(7^k n)$ [21]. A different decomposition technique was used by Fomin and Thilikos [28] to produce an $O(2^{15.13\sqrt{k}} + n^3)$ algorithm, although the authors of [28] warn that their algorithm may not be practical in its presented state. Philip, Raman and Sikdar [53] proved that the restriction of the parameterized Dominating Set to graphs which do not contain $K_{i,j}$ as a subgraph (for a fixed i and j) is fixed-parameter tractable.

1.3 Related Computational Problems

The Set Cover problem takes a universe $\mathcal{U}$, a collection $\mathcal{S}$ of subsets of $\mathcal{U}$ and an integer k and decides whether there exists a collection $S_1, S_2, \ldots, S_k \in \mathcal{S}$ such that $S_1 \cup S_2 \cup \ldots \cup S_k = \mathcal{U}$. The *dimension* of an instance $(\mathcal{U}, \mathcal{S}, k)$ of Set Cover is equal to $|\mathcal{U}| + |\mathcal{S}|$. The associated optimization problem of minimizing k is called Minimum Set Cover. The Set Cover problem is known to be NP-complete [29, 40]. The Dominating Set problem can be reduced to Set Cover with a relatively natural transformation, which meets the criteria for a parameterized reduction and is detailed in Lemma 1.1.

Lemma 1.1 (Karp via Downey and Fellows [21][3]). *Let* (G, k) *be an instance of* Dominating

[3]Karp's original paper [40] does not cover the dominating set problem, but [21] attributes the reduction to Karp.

Set *and let* $n = |V(G)|$. *Taking*

$$\mathcal{U} = V(G), \qquad \mathcal{S} = \{N[v] : v \in V(G)\}$$

results in an instance $(\mathcal{U}, \mathcal{S}, k)$ *of* Set Cover *with dimension* $2n$. *Note that the parameter* k *for the dominating set size is maintained as the size of the cover. This reduction can be performed in polynomial time.*

□

Since reduction in Lemma 1.1 preserves the parameter k between the two problems, it qualifies as a parameterized polynomial time reduction between Dominating Set (parameterized by set size) and Set Cover (parameterized by cover size).

An *integer program* (or *integer linear program*) is an optimization problem on integer variables, consisting of an objective function and a collection of constraints. Integer programming in general is known to be NP-hard, and the special case 0-1 Integer Programming, which takes an integer program and an objective value q and decides whether any solution to the program with objective value at most q exists, is known to be NP-complete [40, 29]. As with Set Cover, there is a straightforward reduction from Dominating Set to 0-1 Integer Programming. For an instance (G, k) of Dominating Set where $|V(G)| = n$ and $V(G) = \{v_0, v_1, \ldots, v_{n-1}\}$, the integer program shown below on n binary variables $x_0, x_1, \ldots, x_{n-1}$ can be used to find dominating sets.

$$\begin{aligned} \text{min.} \quad & x_0 + x_1 + \ldots + x_{n-1} \\ \text{s.t.} \quad & x_i \in \{0, 1\} \text{ for } 0 \leq i \leq n-1 \\ & \text{For each } v_i \in V(G),\ x_i + \sum_{v_i v_j \in E(G)} x_j \geq 1 \end{aligned}$$

Specifically, for any solution to the integer program with objective value k, the set $S = \{v_i :$

$x_i = 1\}$ is a dominating set of size k.

1.4 Algorithms to Compute Minimum Dominating Sets

Previous algorithmic research into the computation of minimum dominating sets for arbitrary graphs has been largely theoretical, and there has been considerable crossover in the techniques used for general exponential time algorithms and the reduction rules used by the fixed-parameter tractability results (such as the algorithm described by Alber et al. for planar graphs [2]). Grandoni, in [32], described an algorithm for finding a minimum dominating set of a graph on n vertices in $O(1.8021^n)$ time and exponential space, with the running time increasing to $O(1.9053^n)$ if only polynomial space is used. The algorithm in [32] was, fundamentally, a wrapper around the reduction from dominating set to minimum set cover given in Lemma 1.1, and the main contribution of [32] was the algorithm for computing minimum set cover, which requires $O(1.3803^n)$ time to solve a set cover instance of dimension n. Since the dominating set problem on a graph G with n vertices can be reduced to a set cover instance of dimension $2n$, the set cover algorithm can solve Dominating Set in $O(1.3803^{2n}) = O(1.9053^n)$ time.

Fomin, Grandoni and Kratsch [27] used an improved analysis technique called 'measure and conquer' to improve the running time on the exponential-space algorithm from [32] to $O(1.5136^n)$ without modifying the algorithm[4]. Van Rooij and Bodlaender [60] used additional reduction rules with the measure and conquer analysis technique to construct a polynomial-space algorithm with running time $O(1.4969^n)$, which was further improved to $O(1.4864^n)$ by Iwata [38]. Over this entire chain of results, there appears to have been no attempt to implement the algorithms or measure their performance in practice, only to produce a theoretical bound (indeed, the only computational results cited appear to be from

[4]The running time is stated as $O(2^{0.598n})$, which is equivalent to $O(1.5136^n)$. For consistency with other results, the latter form is used here.

optimization algorithms for evaluating the bound on the running time). The most complete representation of a dominating solver algorithm among the various articles is given in [60], but consists simply of a list of reduction rules to be applied at each step of a backtracking search. This specification is sufficient for proving the bounds on running time which are established in the article, but likely to be extremely slow in practice, due to the overhead which would accompany evaluating each reduction rule at every step of the search tree. There is very little information in the literature regarding the practical performance of dominating set solvers. Unlike other NP-complete problems, especially the venerable SATISFIABILITY problem, there does not appear to be much research activity into practical general purpose solvers dedicated to the DOMINATING SET problem.

Algorithms for approximating minimum dominating sets have also been studied, particularly *connected dominating sets*, which are dominating sets whose vertices induce a connected subgraph of the original graph. Connected dominating sets have applications to networking, and much of the research into their computation has come from networking fields [63, 66]. Guha and Khuller [33] present an algorithm to find a connected dominating set S of a graph G with n vertices and maximum degree Δ such that the size of S is at most $(H(\Delta)+2)|S^{\min}|$, where H is the harmonic function (which tends asymptotically toward $\log_e(\Delta)$) and $S^{\min}$ is a minimum connected dominating set. Another result in [33] proves that the approximation ratio of their algorithm is close to the best possible under the assumption that P $\neq$ NP.

Chlebík and Chlebíkova [13] (apparently aggregating earlier results) prove that the minimum dominating set and minimum connected dominating set problems cannot be approximated in polynomial time to a ratio of less than $\log_e(n)$ unless NP is a subset of problems solvable in $O(n^{c \log \log(n)})$ time (which is a weaker condition than P = NP). In a similar vein, Irving [37] proved that an approximation algorithm for the minimum independent dominating set cannot achieve an approximation ratio bounded above by any constant. Halldórsson [34] improved the lower bound in [37] and established that the no approximation ratio of

$n^{1-\varepsilon}$ can be achieved unless P = NP. The result in [34] places the independent dominating set problem among the hardest problems in NP as far as approximation is concerned.

Since Dominating Set can be reduced to an instance of various other NP-hard problems, it is possible to find minimum dominating sets with solvers for other NP-complete problems, and it appears that all widely-used general-purpose solvers are based on reductions from Dominating Set to an integer programming problem. The SageMath toolkit [1] contains a fully-featured dominating set solver, with support for independent dominating sets, which uses a Mixed Integer/Linear Programming (MILP) solver to compute its results. SageMath is used later in this thesis as a reference solver for verification and comparison with the algorithms tested in Chapter 4. The integer program used by SageMath for a particular instance of the dominating set problem is the same integer program given in the previous section.

For satisfiability and integer programming problems, there are a considerable variety of solvers available with high performance in practice, even if the underlying algorithms do not achieve the best-known asymptotic bounds on running time. The overarching objective of the research in this thesis is to produce and demonstrate a general purpose solver program for Dominating Set which improves on the performance of the existing integer programming-based solvers for a variety of practical input cases, particularly those for which there exist open problems of interest to researchers. Therefore, one of the objectives of this thesis is a computational tool that will be useful for future domination research. In the process of developing the general purpose solver algorithms, several other open domination problems were solved and new theoretical results were created for some open problems.

Chapter 2 summarizes classes of graphs whose domination parameters are currently studied by researchers and may benefit from computational results. Chapter 3 defines a general framework for a backtracking search algorithm and describes three classes of algorithm based on the framework. Chapter 4 documents a large-scale experimental comparison of the dif-

ferent variants of the framework defined in Chapter 3 against each other and against the SageMath dominating set solver. Chapter 5 describes new results on queen domination problems, including solved cases of the domination number and border domination number of queen graphs, as well as new theoretical results on the border queen problem. Chapter 6 describes a software package containing implementations of the most promising algorithms described in Chapter 3 (based on the experimental data in Chapter 4) and Chapter 7 summarizes the conclusions of this research as well as fruitful avenues for future research.

Chapter 2

Queen Graphs and Other Interesting Graph Classes

This chapter details various families of graphs for which interesting open domination problems exist. Of these families, the queen graphs (Section 2.1) received the most attention in the research for this thesis. The other families, which exhibit substantially different structure from queen graphs, present different challenges for domination algorithms, and therefore were studied primarily as a counterpoint to queen graphs for the purpose of producing a widely useful dominating set solver. Although there many results on the computational complexity of the general Dominating Set problem (and its variants), as detailed in the previous chapter, the computational complexity of the domination problem on the specific families of graphs in this chapter is unknown. Future research may reveal sub-exponential time methods for solving some of these problems, but this thesis assumes that exponential-time algorithms are necessary.

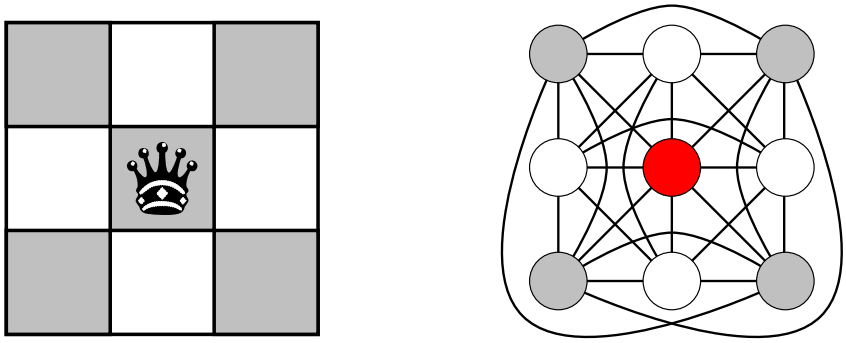
2.1 The Queen Domination Problem

The game of chess has been the source of several interesting mathematical puzzles which are applications of graph theory. For example, the classical knight's tour problem, of finding a sequence of moves for a knight such that every square of the board is visited exactly once, and the final move returns the knight to its starting position, can be considered an instance of the NP-complete Hamiltonian Circuit problem [29].

In chess, a queen may move any distance along a horizontal, vertical, diagonal or back-diagonal line. The famous n-queens problem is to find, for an $n \times n$ chess board, an arrangement of n queens on the board such that no queen can attack another. Although the classical n-queens problem was solved as early as 1874, a variety of related problems and generalizations are still studied [4]. The n-queens problem is a set packing problem. A dual problem is the *queen domination problem*. For a given board size n, the queen domination problem is to find the minimum size of an arrangement of queens such that every square of an $n \times n$ board is either occupied by a queen or can be captured by some queen. As the name suggests, the queen domination problem is an instance of the dominating set problem, and it has interested mathematicians almost as long as the n-queens problem [14].

The *queen graph* on an $n \times n$ board, denoted by $\text{Queen}\,(n)$, contains a vertex for each square of the $n \times n$ chess board and edges between all pairs of vertices in the same row, column, diagonal or back diagonal. The set of squares which can be attacked by a queen on a square $v \in V(\text{Queen}\,(n))$ corresponds to the neighbourhood of v. The vertices of $\text{Queen}\,(n)$ can be numbered $v_{i,j}$ for $0 \leq i, j \leq n-1$ to denote the vertex corresponding to row i, column j of the board, where $v_{0,0},\ v_{n-1,0},\ v_{n-1,n-1}$ and $v_{0,n-1}$ are the bottom-left, top-left, top-right and bottom-right corners of the board, respectively. This numbering allows the vertices of the queen graph to be put into a simple correspondence with the lattice points of the Cartesian plane. The numbering schemes used in previous research vary between authors, with some (such as Sinko and Slater [56]) using the scheme defined here and others (such

as Kearse and Gibbons [42]) placing $v_{0,0}$ at the top left of the board. Figure 2.1 shows the queen graph Queen (3) along with the associated 3×3 chess board. The domination number of Queen (3) is 1, since a queen on the center square can capture all other squares.



(a) A 3×3 chess board (b) The Queen graph Queen (3)

Figure 2.1: A 3×3 chessboard and the associated queen graph Queen (3). A queen in the center square corresponds to a minimum dominating set of the graph.

The value $\gamma(\text{Queen}\,(n))$ is trivially at most $n-2$, since placing queens along the forward diagonal, except in the corner cells, will produce a dominating set. Although previous research has produced upper bounds for the domination number of Queen (n) [10, 11, 64], the exact value of $\gamma(\text{Queen}\,(n))$ is not known exactly for most values of n. For $n \leq 120$, $\gamma(\text{Queen}\,(n)) \leq \lceil n/2 \rceil + 1$ for all open cases [42, 52]. Information on the open cases, as well as a survey of theoretical results, can be found in Östergård and Kaski [52]. Finozhenok and Weakley [26] proved a lower bound on $\gamma(\text{Queen}\,(n))$ which is given in Theorem 2.1.

Theorem 2.1 (Finozhenok and Weakley [26, p. 299])**.** *For all n except $n = 3$ and $n = 11$, $\gamma(\mathit{Queen}\,(n)) \geq \lceil n/2 \rceil$.*

□

Using the algorithms described in Chapter 3, three open cases of the queen domination problem were solved, establishing that $\gamma(\text{Queen}\,(20)) = 11$, $\gamma(\text{Queen}\,(22)) = 12$ and $\gamma(\text{Queen}\,(24)) = 13$. The previous open case $n = 19$ was solved by Kearse and Gibbons

[42] using algorithms developed specifically for queen graphs. Some previous research has studied the application of computational methods to the queen domination problem, notably an article by Fernau from 2010 [25] which surveys various computational approaches and establishes asymptotic upper bounds on the computation time of the queen domination problem, but does not solve any open cases or present evidence that the algorithms are practical. While the algorithms developed in this thesis are designed as general solvers for the dominating set problem, they were created with queen graphs in mind, and the experimental data in Chapter 4 shows that the algorithms introduced here are very effective on queen graphs (and, in general, other dense graphs). Chapter 5 describes the various new results on queen graphs produced by this thesis.

The queen domination problem is part of a large family of related problems on queen graphs [35, 42]. Sections 2.2 and 2.3 describe two of these related problems.

2.2 Irredundant Sets

Let G be a graph on n vertices. An *irredundant set* of the vertices of G is a set $S \subseteq V(G)$ such that, for each $v \in S$, $N[v] - N[S - \{v\}] \neq \emptyset$. Less formally, an irredundant set is one in which, for every vertex v in the set, at least one of the neighbours of v (or v itself) is not dominated by any other vertex in S.

Irredundant sets have a number of connections to dominating sets. The problem of finding an irredundant set is a packing problem, which is a natural dual to covering problems like domination. Lemmas 2.2 and 2.3 establish some fundamental properties of irredundant sets.

Lemma 2.2 (Cockayne [16, p. 463])**.** *Any minimal dominating set is also a maximal irredundant set.*

□

Lemma 2.3 (Cockayne et al. [15, p. 250-251])**.** *If* $\deg(v) \geq 1$ *for all* $v \in V(G)$*, then for all irredundant sets* $X \subseteq V(G)$*, the set* $V(G) - X$ *is a dominating set.*

□

The authors of [15] observe that one consequence of Lemma 2.3, taken together with the result of Lemma 2.2, is that if G has no isolated vertices, then for any minimal dominating set $D \subseteq V(G)$, the set $V(G) - D$ must also be a dominating set. The converse of Lemma 2.2, the statement that a maximal irredundant set is a dominating set, is not true. Figure 2.2 shows a counterexample. The vertices shaded in red represent an irredundant set of size 2. The blue vertex is not dominated by either of the red vertices, so the set is not dominating. However, the set is a maximal irredundant set, since adding any vertex to the set will make one of the red vertices redundant.

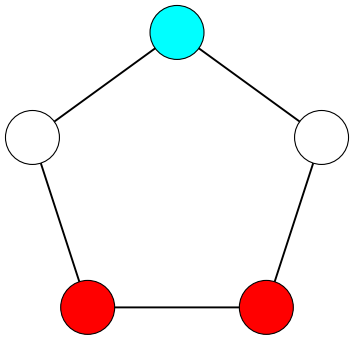


Figure 2.2: A maximal irredundant set (in red) of C_5 which is not a dominating set.

The minimum size of a maximal irredundant set of a graph G, denoted by $\text{ir}(G)$, is called the *lower irredundance number* of G. The maximum size of an irredundant set, denoted by $\text{IR}(G)$, is called the *upper irredundance number*. A consequence of Lemma 2.2 is that $\text{ir}(G) \leq \gamma(G) \leq \text{IR}(G)$ [16].

Lower bounds on $\text{IR}(\text{Queen}\,(n))$ for queen graphs have been well-studied. Currently, the best lower bound (due to Kearse and Gibbons [43, p. 237]) and upper bound (due to Burger,

Cockayne and Mynhardt [10, p. 66]) on IR(Queen (n)) give

$$6n - O(n^{2/3}) \leq \text{IR}(\text{Queen}\,(n)) \leq \left\lfloor 6n + 6 - 8\sqrt{n + 1 + \sqrt{n}} \right\rfloor.$$

Bollobás and Cockayne [6] proved a general lower bound on ir(G), which is given in Lemma 2.4. For the special case of trees, Damaschke [19] proved a lower bound of ir$(G) > \frac{2\gamma(G)}{3}$. Favaron et al. [24] proved lower bounds on both IR(G) and ir(G) for the king's graph (corresponding to the moves of a king in traditional chess). Rautenbach [54] and later Zverovich [67] proved several results on the differences between various graph-theoretic quantities, including domination number, independence number and the irredundance numbers.

Lemma 2.4 (Bollobás and Cockayne [6, p. 198])**.** *If G is a graph on n vertices with maximum degree $\Delta \geq 2$, then ir$(G) \geq \frac{n}{2\Delta - 1}$.*

□

There are families of *irredundance perfect* graphs G for which ir$(G) = \gamma(G)$ [61]. It is unknown whether queen graphs are irredundance perfect, although the very limited data available on ir(Queen (n)) does not contradict this possibility. However, Kearse and Gibbons [42] note that several previous authors have conjectured that ir(Queen (n)) diverges from γ(Queen (n)) eventually. For $n \leq 10$, it is known that ir(Queen (n)) = γ(Queen (n)) [17].

2.3 The Border Queen Problem

The *border queen* problem, proposed by Sinko and Slater [56], is to find a dominating set of Queen (n) using only squares on the border of the chessboard. The border queen problem is an example of a restricted domination problem, in which some vertices must always be excluded from the dominating set. The minimum size of such a set is denoted by bor (Queen (n)). The values of bor (Queen (n)) for $n \leq 13$ were given by Sinko and Slater in

[56]. Chapter 5 describes several new computational and theoretical results for the border queen problem, as well as conjectures for future research to characterize bor (Queen (n)).

In addition to formalizing the border queen problem, [56] also contains constructive proofs of several upper bounds on bor (Queen (n)). A general upper bound originally stated in [56] is given in Theorem 2.8, with a revised proof for consistency with the rest of this section, and a general lower bound is given in Theorem 2.9.

To improve the readability of the proofs in this section, the following terms will be used to describe the various ways that two vertices $v_{i,j}$ and $v_{k,\ell}$ can be neighbours in the queen graph Queen (n).

Condition 2.5. *$v_{k,\ell}$ is a **diagonal neighbour** of $v_{i,j}$ if*

$$k - i = \ell - j$$

Condition 2.6. *$v_{k,\ell}$ is a **back-diagonal neighbour** of $v_{i,j}$ if*

$$k - i = j - \ell$$

Additionally, the following fact will be used to support some of the bounding results later in this section.

Lemma 2.7. *Any border vertex u of Queen (n) has degree $3n - 2$.*

Proof. Without loss of generality, assume that u is a border cell in row 0 (along the bottom of the board). This assumption is valid due to rotational symmetry. There are a total of n vertices in row 0, including u. The column containing u has $n - 1$ vertices, not including u itself. The forward diagonal of u contains one vertex per column, starting at the column to the right of u, and the back diagonal of u contains one vertex per column, starting at column 0 and continuing to the column to the left of u. Together, the two diagonals cover

$n-1$ vertices. □

Theorem 2.8 (Sinko and Slater [56, p. 4824])**.** *For all $n \geq 4$, $bor(Queen(n)) \leq n-2$.*

Proof. Note that the proof given here has been revised from the one given in [56] to use a different construction. Let $n \geq 4$ and let

$$S = \{v_{0,2},\ v_{0,3},\ \ldots,\ v_{0,n-3}\} \cup \{v_{n-1,1},\ v_{n-1,n-2}\}.$$

For visual reference, examples of this construction for $n = 10$ and $n = 11$ are given in Figure 2.3. Since S contains a vertex in every column in the range $1, 2, \ldots, n-2$, every vertex in those columns is dominated. Since S is symmetric about a horizontal reflection, it is sufficient to demonstrate that every vertex in column 0 is dominated to establish that S is a dominating set.

Consider a vertex $v_{i,0}$ where $0 \leq i \leq n-1$. If $i = 0$ or $i = n-1$, $v_{i,0}$ is in a row containing vertices of S and therefore is dominated. If $i = n-2$, then the diagonal neighbour $v_{n-1,1}$ dominates $v_{i,0}$. If $i = 1$, then $v_{i,0}$ is dominated by $v_{n-1,n-2}$, which is a back-diagonal neighbour of $v_{i,0}$. Otherwise, if $2 \leq i \leq n-3$, the vertex $v_{0,i}$, which is a diagonal neighbour of $v_{i,0}$, will dominate $v_{i,0}$.

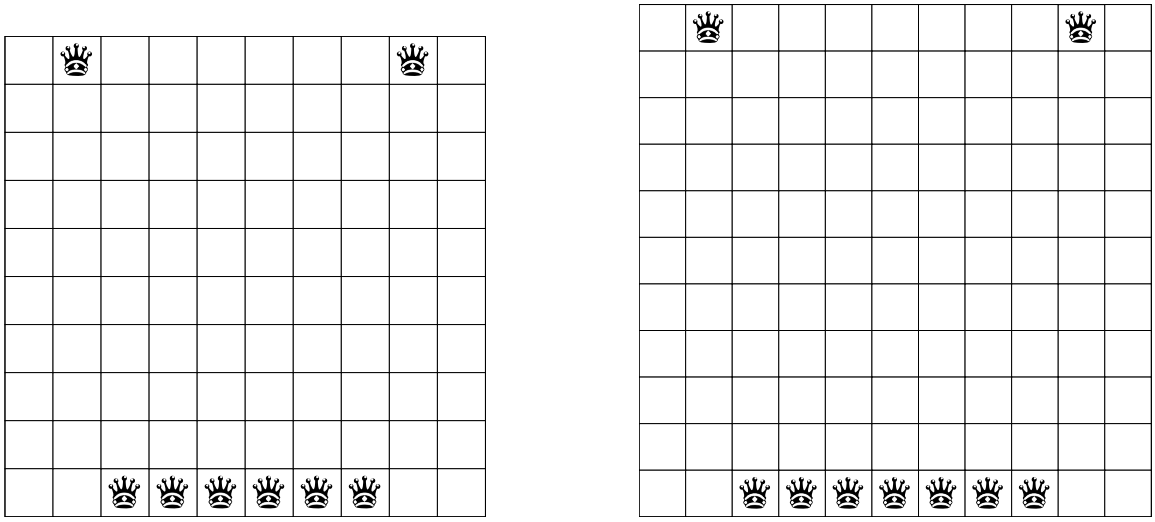
Therefore, S is a dominating set of size $n-2$. □

Theorem 2.9 (Sinko and Slater [56, p. 4824])**.** *For all $n \geq 4$,*

$$bor(Queen(n)) \geq 2n - \frac{1}{2}\sqrt{8n^2 - 40n - 49} - \frac{9}{2}.$$

□

Additional upper bounds are proven in [56] using constructions which leverage the rotational symmetry of the board. Section 5.5 contains a discussion of these results in the context of a new classification of symmetric constructions for the border queen problem.



(a) $n = 10$ (b) $n = 11$

Figure 2.3: Examples of the construction used in the proof of Theorem 2.8.

For a border dominating set S of Queen (n), define $r(S), c(S)$ and $d(S)$ (or, when the context is clear, just r, c and d) to be, respectively, the number of distinct rows, columns and diagonals (both forward and backward) containing vertices in S.

A paper by Burchett [9] contains the statement of a purported formula for a lower bound on bor (Queen (n)). The proof of the bound relies on the assertion that, for a border dominating set S of Queen (n), $r + c + d \leq 3|S| + 3$ (note that the statement of this assertion in the paper differs, due to differing notation). This assertion is false, as evidenced by the border dominating set of Queen (14) in Figure 2.4, which has $|S| = 12$, $r = c = 8$ and $d = 24$, giving $r + c + d = 40 = 3|S| + 4$.

Lemma 2.10 establishes a corrected version of the upper bound on $r + c + d$ in [9]. The incorrect upper bound $r + c + d \leq 3|S| + 3$ was used by Burchett as part of a proof that bor (Queen (n)) $\geq (2n - 5)/3$ [9, p. 181]. The counterexample in Figure 2.4 invalidates the proof of Burchett's bound on bor (Queen (n)), but does not contradict the bound itself (so it may still be possible to prove the bound using other methods). Theorem 2.11 contains a revised bound on bor (Queen (n)) based on Lemma 2.10.

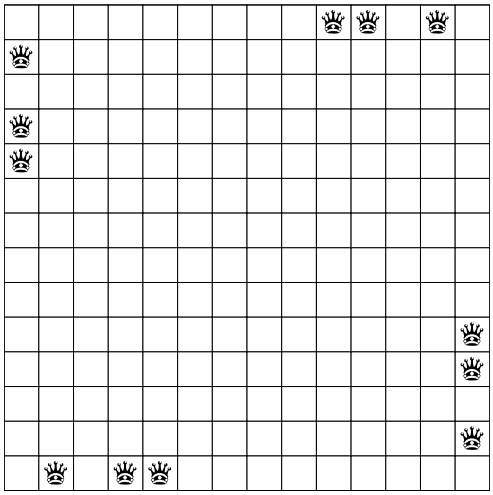


Figure 2.4: Counterexample of an assertion by Burchett in [9].

Lemma 2.10 (Corrected from Burchett [9, p. 181])**.** *If S is a border dominating set of Queen(n), then $r(S) + c(S) + d(S) \leq 3|S| + 4$.*

Proof. Define an auxiliary graph A with vertex set S as follows.

Rule 1: If two vertices $v_{i,j}, v_{k,\ell} \in S$ are diagonal or back-diagonal neighbours, add an edge $v_{i,j}v_{k,\ell}$ to A.

Rule 2: If two vertices $v_{i,j}, v_{k,j} \in S$ are column neighbours and $v_{\ell,j} \notin S$ for all $i < \ell < k$, add an edge $v_{i,j}v_{k,j}$ to A.

Rule 3: If two vertices $v_{i,j}, v_{i,k} \in S$ are row neighbours and $v_{i,\ell} \notin S$ for all $j < \ell < k$, add an edge $v_{i,j}v_{i,k}$ to A.

Vertices in S must lie along one of the four border segments of the board (top, bottom, left, right). By Rule 2 above, all vertices in the same row will be connected by a path, and by Rule 3 above, all vertices in the same column will be connected by a path. Therefore, the graph A can have at most 4 components (since there will be at most one component per border segment). The total number of edges in A is then at least $|V(A)| - 4 = |S| - 4$.

Let m_1, m_2, m_3 count the number of edges created by each of the respective rules above. Then $m_1 + m_2 + m_3 = |E(A)| \geq |S| - 4$. Observe that $r(S) = |S| - m_1, c(S) = |S| - m_2$ and $d(S) = |S| - m_3$. The inequality on m_1, m_2 and m_3 above becomes $r(S) + c(S) + d(S) =$

$4|S| - (m_1 + m_2 + m_3) \leq 4|S| - (|S| - 4) = 3|S| + 4.$ □

Theorem 2.11 (Corrected from Burchett [9, p. 181-182])**.** *For $n \geq 1$, $bor(Queen(n)) \geq \frac{2(n-3)}{3}$.*

Proof. Let S be a minimum border dominating set of Queen (n). By Theorem 2.8, $|S| \leq n-2$. Let i_1 and i_2 be the indices of the first and last rows (respectively) which do not contain a queen, and similarly let j_1, j_2 be the indices of the first and last columns which do not contain a queen. Since $|S| \leq n - 2$, all four indices must exist. In each of rows i_1 and i_2, c vertices are column-dominated by queens in S, and in each of columns j_1 and j_2, r vertices are row-dominated by queens in S. All remaining vertices in rows i_1 and i_2 and columns j_1 and j_2 must be diagonally dominated. The total number of vertices which must be diagonally dominated is

$$2(n - r) + 2(n - c) - 4$$

(where the -4 term accounts for the double-counting of the four squares on the overlap of the rows and columns). Since all such vertices must be diagonally-dominated, and since each of the d diagonals can dominate at most 2 vertices from rows i_1 and i_2 and columns j_1 and j_2,

$$2d \geq 2(n - r) + 2(n - c) - 4$$

giving

$$2(d + r + c) \geq 4n - 4$$

which implies that

$$r + c + d \geq 2n - 2$$

and applying the inequality $r + c + d \leq 3|S| + 4$ from Lemma 2.10 gives

$$\begin{aligned} 3|S| + 4 &\geq 2n - 2 \\ |S| &\geq \frac{2n-6}{3}. \end{aligned}$$

□

2.4 Kneser Graphs

For integers n and k such that $n \geq 1$ and $0 \leq k \leq n$, the *Kneser graph* $\text{Kneser}(n, k)$ has vertices corresponding to k-subsets of $\{1, 2, \ldots, n\}$ and edges between vertices whose corresponding subsets are disjoint [30]. For some combinations of n and k, the domination number $\gamma(\text{Kneser}(n, k))$ has been determined theoretically, but the value of $\gamma(\text{Kneser}(n, k))$ remains an open question in general, and recent results have used computational searches to solve open cases [51]. For $n \leq 4$, the domination number of $\text{Kneser}(n, k)$ is known for all $0 \leq k \leq n$. The smallest open case is currently $\text{Kneser}(12, 5)$ [51].

Table 2.1 summarizes the known bounds and solved cases of the domination number of some Kneser graphs with small n and k. The data in Table 2.1 is transcribed from [51] and [31].

2.5 Covering Codes and Football Pools

Consider a sequence of n football matches. A form of betting on the matches is to place a bet that a given sequence of outcomes (win, lose, or draw) will occur. When all matches are over, the prize money is given to the bettor who predicted all matches correctly. If no such bettor exists, then the money is given to the bettor who made one error, and then the bettor who made two errors, and so on until a winner is found.

Table 2.1: Known bounds on the domination number of Kneser graphs.

n	k			
	2	3	4	5
4	3			
5	3			
6	3	10		
7	3	7		
8	3	7	35	
9	3	7	26	
10	3	6	15	126
11	3	5	15	66
12	3	4	12	37-56
13	3	4	10	23-39
14	3	4	9	16-31
15	3	4	8	15-27
16	3	4	7	12-22
17	3	4	7	11-17
18	3	4	6	11-15
19	3	4	6	11-14
20	3	4	5	11-12
21	3	4	5	11-12

The *Football Pool Problem* is to determine the number of bets needed to ensure that at least one bet will have at most one error [41]. The number of bets needed to guarantee that one bet will be completely accurate is 3^n. To ensure that one bet will have at most one error, the number of bets needed is an open question for $n \geq 6$.

Let Q be an alphabet with $|Q| = q$. For non-negative integers n and r, a *covering code* over Q^n with radius r is a set $C \subseteq Q^n$ such that for all $x = (x_1, x_2, \ldots, x_n) \in Q^n$, there exists some $c = (c_1, c_2, \ldots, c_n) \in C$ such that the Hamming distance $d(x, c)$, which is defined to be

$$d(x, c) = |\{i : x_i \neq c_i\}|,$$

is at most r. For the purpose of finding codes, the alphabet Q can be assumed to be the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$. The minimum size of a covering code of radius r over $\mathbb{Z}_q^n$ is denoted by $K_r(n, q)$. Covering codes have applications in compression, telecommunications

and write-once memories [18]. The dual of a covering code problem is a packing problem, which is equivalent to finding an error-correcting code.

The covering code problem can be stated in graph theoretic terms. For a given n, q and r, define a graph $\mathrm{Code}_r(n, q)$ with vertex set $\mathbb{Z}_q^n$ and edge set

$$E(G_{n,q}) = \{uv : u, v \in V(G) \text{ and } d(u, v) \leq r\}.$$

A covering code of radius r over $\mathbb{Z}_q^n$ is then equivalent to a dominating set of $\mathrm{Code}_r(n, q)$.

Solutions to the football pool problem on n matches correspond to covering codes of radius one over $\mathbb{Z}_3^n$, and the values $K_1(n, 3)$ give the minimum number of bets needed to guarantee that one bet will have at most one error. Brink, in [8], proposed the *Inverse Football Pool Problem*, which asks for the minimum number of bets needed to ensure that one bet will be entirely wrong (that is, the outcome of every match is incorrectly predicted). The inverse football pool problem is equivalent to finding a set $C \subseteq Q^n$ such that for every $x \in Q^n$, there is some $c \in C$ such that $d(x, c) = n$. The size of a minimum solution to the inverse football pool problem on n matches is denoted by $T(n)$ in [8].

Table 2.2 gives the values of $K_1(n, 3)$ and $T(n)$ for small values of n, and the best known bounds for small open cases. The number of vertices in each covering code graph $\mathrm{Code}_1(n, 3)$ is also given (note that the value of n is not equal to the number of vertices). The data for $K_1(n, 3)$ is from [44], which aggregates results on covering codes, including unpublished results. The data for $T(n)$ is taken from [47].

The football pool problem has been the subject of a considerable body of computational research. Among the most recent is the result that $\gamma(\mathrm{Code}_1(6, 3)) \geq 71$ by Linderoth, Margot and Thain [48], who used a specialized integer programming solver to improve the lower bound.

Another important sub-family of the covering code graphs are the *hypercubes*. The hypercube of order n, denoted by Q_n, is defined to be the graph $\mathrm{Code}_1(n, 2)$. Table 2.3 summarizes

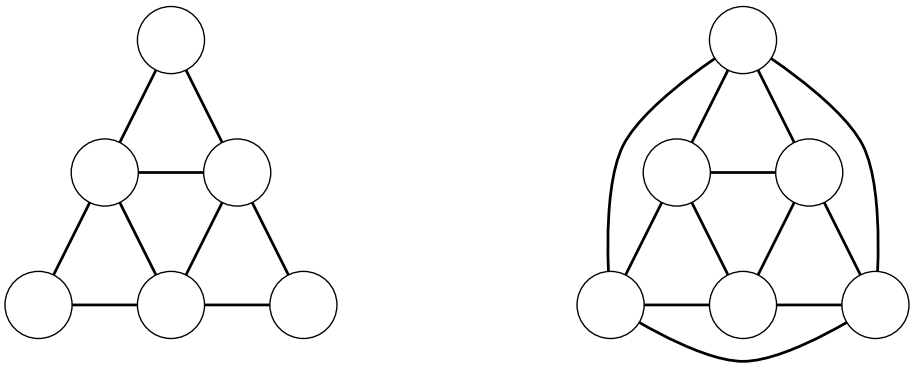
n	# Vertices	Football ($K_1(n,3)$)	Inverse Football ($T(n)$)
1	3	1	2
2	9	3	3
3	27	5	5
4	81	9	8
5	243	27	12
6	729	71-73	18
7	2187	156-186	29
8	6561	402-486	44
9	19683	1060-1269	66-68
10	59049	2854-3645	99-104

Table 2.2: Solutions to the football pool problem and inverse football pool problem on $n = 1, 2, \ldots, 10$.

the known domination numbers of hypercubes, along with the best known bounds for small open cases, with data taken from [44].

n	$\lvert V(Q_n)\rvert$	$\gamma(Q_n)$
1	2	1
2	4	2
3	8	2
4	16	4
5	32	7
6	64	12
7	128	16
8	256	32
9	512	62
10	1024	107-120
11	2048	180-192

Table 2.3: Domination numbers of hypercubes Q_n for $n = 1, \ldots, 11$.



(a) TG (3) (b) HR (3)

Figure 2.5: The triangle grid graph TG (3) and the hex rook graph HR (3).

2.6 Triangle Grid Graphs

For $n \geq 1$, the *triangle grid graph* of order n, denoted TG (n), has vertex set $V(\text{TG}\,(n)) = \{v_{i,j} : 1 \leq i \leq n, 1 \leq j \leq i\}$ and edge set

$$\begin{aligned} E(\text{TG}\,(n)) = \ & \{v_{i,j}v_{i,j+1} : 1 \leq i \leq n, 1 \leq j < i\} \\ & \cup \{v_{i,j}v_{i+1,j} : 1 \leq i < n, 1 \leq j \leq i\} \\ & \cup \{v_{i,j}v_{i+1,j+1} : 1 \leq i < n, 1 \leq j < i\}. \end{aligned}$$

Figure 2.5(a) shows the triangle grid graph of order 3. The definition used here is consistent with Wagon [62] and with the Online Encylopedia of Integer Sequences (OEIS) entry for the domination number of triangular grid graphs [59]. Some other sources use a slightly different numbering scheme starting at $n = 0$. Intuitively, the graph TG (n) is a triangular lattice with side length n.

DeMaio and Tran [20] investigate the domination and independence number of several graphs derived from hexagonal chess. The graph used in [20] to model the behavior of the king in hexagonal chess is identical to a triangle grid graph. Among other results, Demaio

and Tran [20] provide an upper bound

$$\gamma(\mathrm{TG}\,(n)) \leq \left\lceil \frac{n}{7} \right\rceil - 1 + \sum_{i=1}^{\left\lfloor \frac{n}{7} \right\rfloor} 7i - 1$$

on the domination number of $\mathrm{TG}\,(n)$. However, this upper bound is substantially larger than the number of vertices in $\mathrm{TG}\,(n)$ for all values of n. A conjectured closed form for the domination number of $\mathrm{TG}\,(n)$ is given by Wagon [62].

Conjecture 2.12 (Wagon [62])**.** *For $n \geq 14$,*

$$\gamma(\,TG\,(n)) = \left\lfloor \frac{n^2 + 7n - 23}{14} \right\rfloor$$

□

2.7 Hex Rook Graphs

For $n \geq 1$, the *hex rook graph* of order n, denoted by $\mathrm{HR}\,(n)$, comprises a triangle grid graph in which every vertex is adjacent to all vertices in the same horizontal, diagonal or back diagonal line [62]. The term 'hex rook' is due to the graph modelling the possible moves of a rook in hexagonal chess, as defined by Wagon [62]. There are various differing definitions of the moves in hexagonal chess. For example, DeMaio and Tran [20], constrain rooks to move only along horizontal lines, but define a queen in hexagonal chess to move along horizontal, diagonal or back-diagonal lines, equivalent to the rook in Wagon's definition. Figure 2.5(b) shows the hex rook graph of order 3.

The discussion of hex rook graphs in [62] establishes a simple recursive upper bound on $\gamma(\mathrm{HR}\,(n))$, which is given in Lemma 2.13.

Lemma 2.13 (Wagon [62])**.** *For $n \geq 1$, $\gamma(HR\,(n+2)) \leq \gamma(HR\,(n))+1$. Consequently, since*

$\gamma(HR(1)) = \gamma(HR(2)) = 1,$

$$\gamma(HR(n)) \leq \left\lceil \frac{n}{2} \right\rceil .$$

□

The value of $\gamma(\mathrm{HR}(n))$ has been found by computational search for all $n \leq 20$ [62, 58]. Due to the fact that $\gamma(\mathrm{HR}(7)) = 3$ and $\gamma(\mathrm{HR}(n)) = 9$, the recursive formulation of Lemma 2.13 gives an improved upper bound of

$$\gamma(\mathrm{HR}(n)) \leq \left\lfloor \frac{n-1}{2} \right\rfloor$$

for all $n \geq 19$ [62]. Table 2.4 gives the value of $\gamma(\mathrm{HR}(n))$ for $n \leq 24$. The values for $n \geq 21$ are new contributions from this research and were computed with the `unidom` program described in Chapter 6.

Table 2.4: Domination Numbers of Hex Rook Graphs. New results are in red; all other results are from [58].

n	1	2	3	4	5	6	7	8	9	10	11	12
$\gamma(\mathrm{HR}(n))$	1	1	2	2	3	3	3	4	4	5	5	6

n	13	14	15	16	17	18	19	20	21	22	23	24
$\gamma(\mathrm{HR}(n))$	6	7	7	8	8	9	9	9	**10**	**10**	**11**	**11**

2.8 Cartesian Products of Cycles

The *Cartesian product* of two graphs G and H, denoted $G \,\square\, H$, has vertex set $V(G \,\square\, H) = \{(u, v) : u \in V(G), v \in V(H)\}$ and edges between vertices (u_1, v_1) and (u_2, v_2) if either

- $u_1 = u_2$ and $v_1 v_2 \in E(H)$, or
- $v_1 = v_2$ and $u_1 u_2 \in E(G)$.

One famous open problem which connects Cartesian products to dominating sets is Vizing's conjecture [7], which asserts that for any finite graphs G and H, $\gamma(G \,\square\, H) \geq \gamma(G)\gamma(H)$. Klavzar and Seifter [45] studied the domination number of the Cartesian products of cycle graphs, and proved formulas for the domination number of certain products, which are summarized in Theorems 2.14 and 2.15.

Theorem 2.14 (Klavzar and Seifter [45, p. 131-132])**.** *Let $n \geq 4$. Then,*

$$\gamma(C_3 \,\square\, C_n) = n - \lfloor n/4 \rfloor$$
$$\gamma(C_4 \,\square\, C_n) = n.$$

Theorem 2.15 (Klavzar and Seifter [45, p. 133])**.** *Let $n \geq 5$. Then if $n \equiv 0 \pmod 5$,*

$$\gamma(C_5 \,\square\, C_n) = n,$$

and if $n \equiv 1, 2$ or $4 \pmod 5$,

$$\gamma(C_5 \,\square\, C_n) = n + 1.$$

For the experiments in Chapter 4, products of two n-cycles were used in the test dataset. Data on the domination numbers of such graphs is available from the Online Encylopedia of Integer Sequences (OEIS) [57]; the smallest value of n for which $\gamma(C_n \,\square\, C_n)$ is unknown is $n = 22$ [57].

Computational methods for finding dominating sets may be valuable in the study of Vizing's conjecture; as Bresar et al. [7] note, progress on studying possible counterexamples has been slow due to the difficulty of evaluating the domination number of graphs.

Chapter 3

Algorithms

This chapter describes several algorithms which find dominating sets of arbitrary input graphs G. All of the algorithms are based on a common framework to allow meaningful comparison between the recursive structure of the different algorithms without direct comparison of running times. The algorithms can be employed either to find a minimum dominating set of the input graph, with or without initial bounds on the domination number, or to exhaustively generate all dominating sets of a given size.

Section 3.1 describes the common framework used by all of the algorithms in the rest of the chapter. The framework uses a recursive backtracking search, which has running time exponential in the number of vertices in the worst case. Section 3.2 describes an implementation of the framework using a simple bounding condition based on the maximum degree of the graph. Section 3.3 describes an implementation which uses a dynamic bounding condition based on the number of undominated vertices which can be dominated by each vertex. Section 3.4 describes an implementation using a different bounding condition which was derived by examining the properties of dominating sets of queen graphs.

3.1 Backtracking Framework

Since the goal of this research was to compare different algorithmic approaches to finding minimum dominating sets, it was necessary to define a common framework for all of the tested algorithms to allow for a meaningful comparison. In practice, the performance of an algorithm for a large-scale combinatorial search would likely be measured by its running time or memory consumption. However, metrics like running time are sensitive to implementation-specific factors, such as the choice of data structures, and overhead introduced by the programmer's chosen code structure. Although the particulars of the implementation are important, the asymptotic behavior of the algorithm tends to depend more on the size of the search tree, and algorithms which produce smaller search trees tend to be more viable in the general case. Moreover, once an algorithm which produces the smallest-possible search tree is found, implementation-specific details can be optimized to improve its speed further.

To allow comparison between different backtracking algorithms, a common framework algorithm was designed, to allow direct comparison between the search tree sizes for different methods. The framework specifies a recursive procedure to build a dominating set of a graph by considering several choices for adding one vertex to the set at each recursive step, and terminating branches of the tree which will never produce a smaller dominating set than the best candidate seen so far. The recursive component of the algorithm is contained in the FindDominatingSet procedure in Framework 3.1. The parameters to FindDominatingSet are

- The input graph G.
- A partial dominating set P.
- A set $C \subseteq V(G)$ of candidate vertices.
- The best dominating set B found so far.

- An upper bound `desired_size` on the size of the dominating set to find. If no set of size `desired_size` or smaller can be found, recursion returns.

For the initial call to FindDominatingSet, set $P = \emptyset, C = V(G), B = V(G)$. To find a minimum dominating set of the graph, `desired_size` is set to $|V(G)|$. To decide if G has a dominating set of size at most k, `desired_size` is set to k.

Framework 3.1 Framework for a Backtracking Search

```
1: procedure FindDominatingSet(G, P, C, B, desired_size)
2:     if P is a dominating set then
3:         if |P| < |B| then
4:             Overwrite B with a copy of P
5:         end if
6:         return
7:     end if
8:     Compute a lower bound k on the size of a dominating set D
9:     such that P ⊆ D ⊆ P ∪ C.
10:    if k ≥ |B| or k > desired_size then return
11:    T ← ∅
12:    v ← An undominated vertex of G
13:    for each vertex u ∈ N[v] ∩ C do
14:        T ← T ∪ {u}
15:        FindDominatingSet(G, P ∪ {u}, C − T, B, desired_size)
16:    end for
17: end procedure
```

The main differences between the implementations studied in Sections 3.2 - 3.4 are contained in the three unspecified aspects of the framework, which affect the size of the search tree in different ways. There is a significant interplay between the three aspects, since the bounding condition (line 10) may rely on the set of candidate vertices C to determine the minimum size of a dominating set containing the current partial set P, and the set C may in turn be affected by the choice of vertex to dominate (line 12) and the order in which potential dominators are tried (in the loop on line 13).

3.1.1 Bounding Condition

The conditional on line 10 of the framework will terminate the branch of recursion if a lower bound k on the size of a dominating set D containing the current partial set P is too large to be a possible minimum. The framework does not describe how the bound k should be computed. Since the conditional on line 10 is the only means by which the branch can be terminated early, the bounding condition is the most critical component to optimizing the size of the search tree. The purpose of the other two unspecified aspects of the framework, described in Sections 3.1.2 and 3.1.3, is to increase the likelihood that the bounding condition on line 10 fails and recursion returns.

3.1.2 Vertex Selection

At each recursive step, if the set P is not a dominating set, at least one vertex must be added to P, and there must be at least one vertex $v \in V(G)$ which is not dominated by a vertex in P. The framework specifies that a particular undominated vertex v must be chosen on line 12, and then dominated by adding one of its neighbours (or itself) to P before continuing recursion. The choice of v can influence the ability of the algorithm to find a small set quickly, and also affect whether the bounding conditions fail at a future recursive step.

3.1.3 Neighbour Ordering

After an undominated vertex v has been chosen, each of the candidate vertices in $N[v]$ is tried as a dominator for v. Once a vertex $u \in N[v]$ has been tried and the corresponding recursive call returns, u ceases to be a candidate vertex for all future recursive calls below the current level (the set T in the framework contains all vertices that have been excluded as candidates). The order in which the vertices in $N[v]$ are tried is not specified by the

framework, but can have a significant impact on the size of the search tree of the algorithm. If a small dominating set is found on the first iteration of the loop on line 13, the recursive branches created by future iterations may be able to terminate early. Additionally, with each successive iteration of the loop, more candidate vertices are excluded, increasing the chance that no viable dominating set can be found.

3.2 Bounding With Fixed Vertex Ordering

Bounding Strategy 3.1 gives a basic bounding condition for line 10 of Framework 3.1.

Bounding Strategy 3.1. *Let G be a graph on n vertices and let Δ be the maximum degree of a vertex of G. If $P \subseteq V(G)$ is a partial dominating set and $|N[P]| = q$, then any dominating set D such that $P \subseteq D$ will have size at least*

$$k = |P| + \frac{n - q}{\Delta + 1}.$$

To use Bounding Strategy 3.1, it is necessary to track both the size of P and the size of $N[P]$. The maximum degree Δ can be precomputed. Since Framework 3.1 stores both $|P|$ and $|N[P]|$ for other reasons (to track whether all vertices are dominated), the bound can be evaluated very efficiently. Computing the value of k requires a constant number of operations, and updates to the values of $|P|$ and $|N[P]|$ require a constant number of operations for each modification (addition/removal of a vertex) to P.

3.2.1 Implementation: Algorithm 3.2

Algorithm 3.2 contains pseudocode for an implementation of Framework 3.1 using Bounding Strategy 3.1 as the bounding condition. The algorithm uses the following data structures.

- The input graph G is represented by an adjacency list structure, with the number of

vertices n and degree of each vertex stored separately. The vertices of the graph are indexed starting at 0 and numbered $v_0, v_1, v_2, \ldots, v_{n-1}$.

- The partial dominating set P and best known dominating set B are represented by array-based lists. The size of each list is tracked separately (so the size can be used in $O(1)$ time).

- The value of Δ is pre-computed and stored in an array.

- The set C of candidate vertices and the set $N[P]$ which contains all dominated vertices are implemented as boolean arrays of n elements, where element i is `true` if and only if vertex i is in the set.

Algorithm 3.2 chooses the lowest numbered undominated vertex v_i to dominate at each step. To dominate v_i, each element of $N[v_i] \cap C$ is considered as a dominator, in three groups: first, v_i itself is tried, then the undominated neighbours of v_i, followed by the neighbours of v_i which are already dominated.

Undominated neighbours are iterated through before dominated neighbours as a heuristic, to maximize the chance that the chosen dominator also dominates other vertices besides v_i. Within each group, neighbours are iterated through in reverse numerical order, again as a heuristic, to take advantage of preprocessing which renumbers the vertices of the graph. For example, if the vertices of the graph are numbered in ascending order of degree, then lower degree vertices are chosen for domination first, while higher degree vertices are tried first as dominators.

The recursive FindDominatingSet procedure in Algorithm 3.2 has all of the parameters of the version in Framework 3.1, as well as the index i of the lowest numbered vertex that has not been dominated by a previous level of recursion. Note that v_i may be dominated anyway (as a side effect of an earlier addition to P), so Algorithm 3.2 may advance the value of i to find an undominated vertex (this occurs in the loop on lines 9 - 11 of the pseudocode).

Algorithm 3.2 Backtracking Algorithm using Bounding Strategy 3.1

```
1: procedure FindDominatingSet(G, P, C, B, desired_size, i)
2:     n ← |V(G)|
3:     if |N[P]| = n then
4:         if |P| < |B| then
5:             Overwrite B with a copy of P
6:         end if
7:         return
8:     end if
9:     while v_i ∈ N[P] do
10:        i ← i + 1
11:    end while
12:    {Compute Bounding Strategy 3.1}
13:    k ← |P| + (n − |N[P]|)/(Δ+1)
14:    if k ≥ |B| or k > desired_size then return
15:    F ← Empty stack
16:    {Try vertex v_i, if applicable}
17:    if v_i ∈ C then
18:        Remove v_i from C
19:        FindDominatingSet(G, P ∪ {v_i}, C, B, desired_size, i + 1)
20:        Push i onto F
21:    end if
22:    {Try undominated neighbours of v_i}
23:    for each neighbour v_j ∈ N[v_i] do
24:        if v_j ∈ C and i ≠ j and v_j ∉ N[P] then
25:            Remove v_j from C
26:            FindDominatingSet(G, P ∪ {v_j}, C, B, desired_size, i + 1)
27:            Push j onto F
28:        end if
29:    end for
30:    {Try dominated neighbours of v_i}
31:    for each neighbour v_j ∈ N[v_i] do
32:        if v_j ∈ C and i ≠ j and v_j ∈ N[P] then
33:            Remove v_j from C
34:            FindDominatingSet(G, P ∪ {v_j}, C, B, desired_size, i + 1)
35:            Push j onto F
36:        end if
37:    end for
38:    while F is non-empty do
39:        j ← Pop(F)
40:        Add v_j to C.
41:    end while
42: end procedure
```

For the initial call to FindDominatingSet, i is set to zero and the other parameters are set as indicated in Section 3.1 for the framework. Since the numbering of vertices is determined by the structure of the adjacency list for the input graph G, any renumbering of the graph is done before the FindDominatingSet procedure is initially called. Three different vertex ordering heuristics were evaluated in the experimental evaluation of Algorithm 3.2 in Section 4.3.

3.3 Domination Degree Algorithms

The neighbourhood $N[P]$ of a partial dominating set P contains all vertices dominated by some vertex of P. For a vertex $v \in V(G) - P$, define the *domination degree* of v with respect to P, denoted by $\mathrm{DD}_P(v)$, to be the number of vertices in $N[v]$ which are not dominated by a vertex in P.

If P is not a dominating set of G, a lower bound on the size of a dominating set containing every vertex in P can be constructed using the domination degree of all available vertices. Observation 3.2 relates the domination degree of the vertices in $V(G) - P$ to the size of a dominating set created from P, and Bounding Strategy 3.3 formalizes the bound itself.

Observation 3.2. *Let G be a graph on n vertices, and let C be a set of candidate vertices for augmenting a partial dominating set P. Consider a dominating set D such that $P \subseteq D \subseteq P \cup C$. Then,*

$$\sum_{v \in D-P} \mathrm{DD}_P(v) \geq |V(G) - N[P]|.$$

□

Bounding Strategy 3.3. *Let G be a graph on n vertices, and let C be a set of candidate vertices for augmenting a partial dominating set P. Let k be the number of vertices which are not dominated by P. Assume that the vertices of C are ordered $u_1, u_2, ...u_{|C|}$ such that $\mathrm{DD}_P(u_i) \geq \mathrm{DD}_P(u_{i+1})$ for $i = 1, 2, \ldots, |C|$.*

5	10	♛	9	8	9	7	5
12	9	10	10	12	10	11	6
12	13	6	13	11	13	9	9
11	11	13	11	14	10	12	7
10	13	10	15	10	13	9	9
13	12	13	11	15	10	12	5
11	14	9	13	10	13	9	9
10	8	6	7	9	7	9	♛

Figure 3.1: An 8×8 board with two queens.

Then any dominating set D such that $P \subseteq D \subseteq P \cup C$ must contain at least $|P| + q$ vertices where q is the smallest integer such that

$$\sum_{i=1}^{q} \mathrm{DD}_P(u_i) \geq k.$$

Figure 3.1 shows a sample configuration of an 8×8 board with two queens. Squares which are dominated by either queen are highlighted in blue. For each square that does not contain a queen, the domination degree is given in the center of the cell (cells containing queens have a domination degree of zero). Since the board contains 26 undominated cells, and since there are two cells with domination degree 15, Theorem 3.3 implies that a minimum of two queens are needed to complete the dominating set (and, consequently, any dominating set containing those two queens must contain at least 4 queens).

For an undominated vertex $v \in V(G) - N[P]$, define the *candidate degree* of v with respect to C, denoted by $\mathrm{CD}_C(v)$, to be the number of neighbours in $N[v]$ which are members of the candidate set C. Lemmas 3.4 and 3.5 formalize two simple properties of the candidate degree of a vertex.

Lemma 3.4. *Let G be a graph on n vertices and let $C \subseteq V(G)$ be a set of candidate vertices*

for augmenting a partial dominating set $P \subseteq V(G)$. If there exists an undominated vertex $v \in V(G) - N[P]$ with $\mathrm{CD}_C(v) = 0$, then there does not exist a dominating set D of G such that $D \subseteq P \cup C$.

□

Lemma 3.5. *Let G be a graph on n vertices and let $C \subseteq V(G)$ be a set of candidate vertices for augmenting a partial dominating set $P \subseteq V(G)$. Suppose $v \in V(G) - N[P]$ is an undominated vertex such that $\mathrm{CD}_C(v) = 1$, and let w be the lone element of $N[v] \cap C$. Then, in any dominating set $D \subseteq P \cup C$, $w \in D$.*

□

Lemma 3.4 implies a new termination condition for the recursive search: if any vertex has candidate degree zero, recursion can return, since no dominating set can be produced. Lemma 3.5 gives a condition in which a vertex is forced to be added to the partial set P to avert the stopping condition in Lemma 3.4. Intuitively, the candidate degree of an undominated vertex v corresponds to the number of ways that v could be dominated by a finished dominating set, and vertices with low candidate degrees are 'at risk' of having no viable dominators. Therefore, candidate degree information can be incorporated into the vertex selection rule of the backtracking algorithm (line 12 of Framework 3.1).

To use Bounding Strategy 3.3 for the bounding condition on line 10 of Framework 3.1, it is necessary to know the multiset of domination degrees of all vertices in the graph. Similarly, to use minimum or maximum candidate degree to choose an undominated vertex to dominate, it is necessary to compute the candidate degree of each vertex. Both candidate degrees and domination degrees may change between recursive calls, so it is necessary to recompute at least some of the information as the algorithm progresses. One option is to completely recompute all values at each step (for example, by iterating over all vertices to compute their domination degree). Another option is to use a persistent data structure and make selective

updates as values change. Although the latter option may be intuitively appealing (since unnecessary work seems to be avoided), the extra overhead of a complicated data structure may outweigh the cost of the unnecessary recomputations when the recomputation is done by a simple iterative process, especially in cases where a large number of values change between steps.

For the algorithm described in this section, two data structures were used to maintain the multiset of domination degrees and candidate degrees. Both data structures allow only changed values to be updated at each step, but also try to minimize practical overhead to compete with recomputation-based approaches. This research project did not profile the performance of the recomputation-based method, and therefore there is no experimental evidence to suggest that recomputation is inferior to the data structures used here. Experimentally evaluating the various techniques is a topic for future research.

Section 3.3.1 describes the data structure used to maintain the multiset of domination degrees. The data structure used for candidate degrees, which could be considered an extension of the domination degree structure, is described in Section 3.3.2.

3.3.1 Domination Degree Multiset

With respect to a partial dominating set $P \subseteq V(G)$, the domination degrees of the vertices of a graph G on n vertices with maximum degree Δ will always lie in the range $[0, \Delta + 1]$. To evaluate Bounding Strategy 3.3, it is necessary to maintain at least a partial ranking of the domination degrees of the candidate vertices. The data structure described in this section maintains a complete ranking of the domination degrees of all candidate vertices, allowing the bound to be evaluated by iterating over the domination degrees in descending order until the conditions of Bounding Strategy 3.3 have been met. Since the data structure must be able to quickly update the ranking, it is also designed for the following operations to be possible with little overhead.

- For any vertex v_i, the domination degree $\mathrm{DD}_P(v_i)$ can be computed with a simple lookup.
- When $\mathrm{DD}_P(v_i)$ changes, the ranked list can be updated by making local modifications to a linked list.
- When a vertex $v_i \in C$ is removed from the candidate set C, its domination degree can be easily removed from the ranked list.
- Similarly, when a vertex $v_i \notin C$ is added to the candidate set C, its domination degree can be easily added to the ranked list.

The data structure is based on a doubly linked list, and represents the multiset of domination degrees of vertices in the candidate set C. Any multiset M can be represented by a set S of pairs (d, c) where $d \in M$ and c is the multiplicity of d in M. Each node of the linked list contains one (d, c) pair, with the elements of the list sorted in ascending order by the value of d (the domination degree)[1].

In practice, linked lists are useful data structures to represent sequences which require fast iteration, insertion and removal of nodes, but not necessarily random access. By contrast, an array (or vector) allows fast iteration and random access, but has worst-case linear running time for insertion and removal of elements. Although linked list operations such as insertion and removal may require constant time, the overhead associated with managing the list (such as rearranging pointers and managing memory) can negate the advantages of using the list for small sequences. As discussed in Chapter 1, in the context of this research project, whether or not a technique is asymptotically optimal is not important if the technique has unreasonably high overhead.

To address the performance issues that arise in practice for linked lists, the data structure described in this section uses several optimizations to reduce overhead. First, dynamic

[1]The choice of ascending vs. descending order did not have any effect on the performance of the data structure, since the list was doubly linked and therefore symmetric.

memory allocation (the `malloc` function in C and the `new` operator in C++) is not used. Instead, the set of nodes is static, and only their contents and interconnections change over the course of the recursive dominating set computation. Since the number of possible domination degrees is bounded above by the value $\Delta + 1$, all of the nodes can be allocated in advance. Second, to streamline the process of inserting and removing nodes, a circular list is used, with a sentinel node representing the "beginning" and "end" (the `next` pointer on the sentinel node refers to the first element of the list and the `previous` pointer of the sentinel node refers to the last element of the list). Using a sentinel node instead of the standard `front` and `back` (or `head` and `tail`) pointers allows the same simple code to be used for all insertions and removals, instead of having special cases for inserting or removing at the beginning or end. In the pseudocode listings later in this section, the value SENTINEL is used to refer to the sentinel node.

Each node in the list contains the following data elements.

- `next` and `previous`: Pointers to the next and previous nodes in the list (respectively).
- `domination_degree`: The domination degree value associated with the node.
- `count`: The number of vertices with this domination degree.
- `candidate_count`: The number of candidate vertices with this domination degree.

Since the node corresponding to the domination degree of a particular vertex is often needed (for example, when the domination degree of that vertex changes), the data structure also maintains an array `VertexDegreeNode` of n pointers, with `VertexDegreeNode[i]` pointing to the domination degree node for vertex v_i. This also allows the domination degree of a vertex to be looked up quickly. Similarly, since the node corresponding to a particular domination degree may be needed (for example, when adjusting the domination degree of a vertex), an array `DegreeNodes` of $\Delta + 1$ pointers is also maintained, with `DegreeNodes[i]` pointing to the node for domination degree i.

For the purposes of Bounding Strategy 3.3, only vertices which are in the candidate set C are needed. Since a vertex may be removed from C (thereby removing it from consideration of the bound) and then later reinserted into C, the data structure continues to track the domination degree of all vertices even when they are removed from C, but does not include non-candidate vertices in the `count` field of each node.

The data structure defines the following operations.

Operation	**Specification**
INIT(G)	Initialize the data structure for the graph G, with the domination degree of each vertex v_i set to $\deg(v_i)+1$ and every vertex assumed to be included in C.
ADDCANDIDATE(v_i)	Called when vertex v_i is added to the candidate set C.
REMOVECANDIDATE(v_i)	Called when vertex v_i is removed from the candidate set C.
DOMINATIONDEGREE(v_i)	Return the domination degree of the vertex v.
INCREMENT(v, C)	Increase the domination degree of v by 1.
DECREMENT(v, C)	Decrease the domination degree of v by 1.
MINTODOMINATE(k)	Evaluate Bounding Strategy 3.3 and return the minimum number q of vertices needed to dominate k vertices.

Pseudocode for each operation (except INIT, which is only executed once) is given in Algorithm 3.3. For clarity, the procedures in the pseudocode use the values SENTINEL (referring to the sentinel node), `VertexDegreeNode` and `DegreeNodes` as global variables. The pseudocode uses C-style notation to denote pointer dereferencing.

3.3.2 Candidate Degree Priority Queue

Algorithm 3.5 in the next section chooses an undominated vertex to dominate at each step based on candidate degree (either a vertex with minimum or maximum candidate degree, depending on the particular implementation). A vertex with minimum (or maximum) candidate degree can be found at each recursive step by iterating over all candidate vertices.

Algorithm 3.3 Operations of the Domination Degree Multiset Structure

```
1: procedure AddCandidate(v_i)
2:     Increment VertexDegreeNode[i]->candidate_count
3: end procedure

4: procedure RemoveCandidate(v_i)
5:     Decrement VertexDegreeNode[i]->candidate_count
6: end procedure

7: procedure DominationDegree(v_i)
8:     return VertexDegreeNode[i]->domination_degree
9: end procedure

10: procedure MinToDominate(k)
11:     q ← 0
12:     {Iterate backwards from the largest domination degree}
13:     node ← Sentinel.previous
14:     while node ≠ Sentinel do
15:         {Compute the maximum number c of vertices which can be dominated by vertices}
16:         {with the domination degree in the current node}
17:         c ← (node->domination_degree)·(node->candidate_count)
18:         if k ≤ c then
19:             return q + ⌈k/node->domination_degree⌉
20:         else
21:             q ← q+ node->candidate_count
22:             k ← k− c
23:         end if
24:         node ← node->previous
25:     end while
26:     {If the Return statement above was not reached, it is not possible to dominate}
27:     {the requested number of vertices.}
28:     return ∞
29: end procedure
```

```
30: procedure INCREMENT(v_i, C)
31:     old_node ← VertexDegreeNode[i]
32:     old_deg ← old_node->domination_degree
33:     new_deg ← old_deg+1
34:     new_node ← DegreeNodes[new_deg]
35:     old_count ← old_node->count
36:     new_count ← new_node->count
37:     if new_count = 0 then
38:         new_node->next ← old_node->next
39:         new_node->previous ← old_node
40:         old_node->next->previous ← new_node
41:         old_node->next ← new_node
42:     end if
43:     VertexDegreeNode[i] ← new_node
44:     Decrement old_node->count
45:     Increment new_node->count
46:     if v_i ∈ C then
47:         Decrement old_node->candidate_count
48:         Increment new_node->candidate_count
49:     end if
50:     if old_count = 0 then
51:         new_node->previous ← old_node->previous
52:         new_node->previous->next ← new_node
53:         old_node->next ← NULL
54:         old_node->previous ← NULL
55:     end if
56: end procedure
```

```
57: procedure DECREMENT(v_i, C)
58:     old_node ← VertexDegreeNode[i]
59:     old_deg ← old_node->domination_degree
60:     new_deg ← old_deg-1
61:     new_node ← DegreeNodes[new_deg]
62:     old_count ← old_node->count
63:     new_count ← new_node->count
64:     if new_count = 0 then
65:         new_node->next ← old_node
66:         new_node->previous ← old_node->previous
67:         old_node->previous->next ← new_node
68:         old_node->previous ← new_node
69:     end if
70:     VertexDegreeNode[i] ← new_node
71:     Decrement old_node->count
72:     Increment new_node->count
73:     if v_i ∈ C then
74:         Decrement old_node->candidate_count
75:         Increment new_node->candidate_count
76:     end if
77:     if old_count = 0 then
78:         new_node->next ← old_node->next
79:         new_node->next->previous ← new_node
80:         old_node->next ← NULL
81:         old_node->previous ← NULL
82:     end if
83: end procedure
```

This approach may be the fastest in some cases, either because the number of candidate vertices is small or because of the low overhead of a simple iterative loop. An alternative approach uses a data structure to track the candidate degree of each vertex as it changes over the course of the algorithm, to allow vertices with minimum (or maximum) candidate degree to be found without examining all candidate vertices. In particular, the requirements of this problem align with a data structure called a *priority queue*.

A priority queue stores a collection of objects, each with a numerical parameter[2] called a *key*, and has two fundamental operations: Insert, which adds an element to the collection and RemoveMin (or RemoveMax, although generally priority queues only allow removal of minimum or maximum, not both) which removes and returns the element with the smallest (or largest) key. A third operation, Adjust, which changes the key of some element already in the collection, may also be defined.

When the objects in the priority queue correspond to vertices in a graph (and are indexed $0, 1, \ldots, n$, using a heap to implement the priority queue results in all three operations requiring $O(\log n)$ time: an array-based heap is used, combined with a table that maps each vertex index to its current index in the heap (to allow easy adjustment of keys). However, although the asymptotic running time of the heap operations is efficient, the heap operations are relatively high-overhead (and in particular, require a large number of conditional branches, which are likely to slow down modern processors). Heaps are not the only option for implementing priority queues, and for this task, a linked-list based priority queue was designed, where most operations require constant time and have relatively low overhead.

The domination degree multiset structure in Section 3.3.1 maintains the domination degrees of all vertices in the graph in ascending order, to allow Bounding Strategy 3.3 to be computed quickly. It is not necessary to maintain any kind of ranking of vertices to be able to choose vertices with minimum or maximum candidate degree. However, the

[2]In general, any totally ordered quantity can be used (such as strings under lexicographical order), but for the purposes of this section, numerical keys are assumed.

domination degree multiset structure can be adapted to implement a priority queue which tracks candidate degree efficiently.

The candidate degree priority queue is a linked list multiset data structure (tracking candidate degree instead of domination degree) in which each node of the list corresponds to a particular candidate degree and maintains a count of how many vertices currently have that candidate degree, as well as a set of all undominated vertices with that candidate degree. The set of vertices is represented by a circular linked list (in which a sentinel node marks the beginning and end of the list). As in the domination degree structure, the list is in ascending order (with the lowest candidate degree at the first node).

Each node in the candidate degree list (a `CandidateDegreeNode`) contains the following data elements.

- `next` and `previous`: Pointers to the next and previous nodes in the list (respectively).
- `candidate_degree`: The candidate degree value associated with the node.
- `count`: The number of vertices with this candidate degree.
- `undominated_count`: The number of undominated vertices with this candidate degree.
- `undominated_list_sentinel`: Sentinel node for the list of undominated vertices.

The nodes in the undominated list for each candidate degree (`VertexNode`) have the following data elements.

- `next` and `previous`: Pointers to the next and previous nodes in the list (respectively).
- `index`: The index of the vertex corresponding to this node.
- `degree_node`: A pointer to the `CandidateDegreeNode` currently associated with this vertex.
- `is_dominated`: Boolean value indicating whether this vertex is currently dominated.

As in the domination degree multiset, several arrays of pointers are maintained to allow easy lookup.

- `VertexNodes`: An array of n pointers, mapping each vertex index to its unique `VertexNode`
- `DegreeNodes`: An array of $\Delta + 1$ pointers, mapping each candidate degree value to the corresponding candidate degree node.

The candidate degree priority queue data structure defines the following operations. Note that the GetMinUndominated and GetMaxUndominated find and return vertices with minimum or maximum candidate degree, but do not remove these vertices from the candidate set.

Operation	**Specification**
Init(G)	Initialize the data structure for the graph G, with the candidate degree of each vertex v_i set to $\deg(v_i)+1$ and every vertex assumed to be included in C.
Dominate(v_i)	Called when vertex v_i is dominated (and was not previously dominated).
Undominate(v_i)	Called when vertex v_i is no longer dominated (and was previously dominated).
CandidateDegree(v_i)	Return the candidate degree of the vertex v_i.
Increment(v_i, C)	Increase the candidate degree of v_i by 1.
Decrement(v_i, C)	Decrease the candidate degree of v_i by 1.
GetMinUndominated()	Return the index of an undominated vertex with minimum candidate degree, or -1 if no such vertex exists.
GetMaxUndominated()	Return the index of an undominated vertex with maximum candidate degree, or -1 if no such vertex exists.

Pseudocode for each operation (except Init, which is only executed once) is given in Algorithm 3.4. Two helper functions SpliceIn and SpliceOut, which manage the insertion and removal (respectively) of nodes into the lists of undominated vertices, are also given.

Algorithm 3.4 Operations of the Candidate Degree Priority Queue Structure

```
1: procedure CandidateDegree(v_i)
2:     return VertexNodes[i]->degree_node->candidate_degree
3: end procedure

4: procedure SpliceIn(vertex_node)
5:     degree_node ← vertex_node->degree_node
6:     {Insert the vertex node at the end of the list of undominated vertices}
7:     vertex_node->next ← degree_node->undominated_list_sentinel
8:     vertex_node->previous ← degree_node->undominated_list_sentinel->previous
9:     vertex_node->next->previous ← vertex_node
10:    vertex_node->previous->next ← vertex_node
11: end procedure

12: procedure SpliceOut(vertex_node)
13:    vertex_node->next->previous ← vertex_node->previous
14:    vertex_node->previous->next ← vertex_node->next
15:    vertex_node->next ← NULL
16:    vertex_node->previous ← NULL
17: end procedure

18: procedure Dominate(v_i)
19:    vertex_node ← VertexNodes[i]
20:    vertex_node->is_dominated ← True
21:    Decrement vertex_node->degree_node->undominated_count
22:    SpliceOut(vertex_node)
23: end procedure

24: procedure Undominate(v_i)
25:    vertex_node ← VertexNodes[i]
26:    vertex_node->is_dominated ← True
27:    Increment vertex_node->degree_node->undominated_count
28:    SpliceIn(vertex_node)
29: end procedure
```

```
30: procedure INCREMENT(v_i, C)
31:     vertex_node ← VertexNodes[i]
32:     old_degree_node ← vertex_node->degree_node
33:     old_deg ← old_degree_node->domination_degree
34:     new_deg ← old_deg+1
35:     new_degree_node ← DegreeNodes[new_deg]
36:     old_count ← old_degree_node->count
37:     new_count ← new_degree_node->count
38:     if new_count = 0 then
39:         new_degree_node->next ← old_degree_node->next
40:         new_degree_node->previous ← old_degree_node
41:         old_degree_node->next->previous ← new_degree_node
42:         old_degree_node->next ← new_degree_node
43:     end if
44:     Decrement old_degree_node->count
45:     if vertex_node->is_dominated = False then
46:         {If this vertex is undominated,}
47:         {remove it from the undominated list of the old degree node}
48:         SPLICEOUT(vertex_node)
49:         Decrement old_degree_node->undominated_count
50:     end if
51:     vertex_node->degree_node ← new_degree_node
52:     if vertex_node->is_dominated = False then
53:         {If this vertex is undominated,}
54:         {add it to the undominated list of the new degree node}
55:         SPLICEIN(vertex_node)
56:         Increment new_degree_node->undominated_count
57:     end if
58:     Increment new_degree_node->count
59:     if old_count = 0 then
60:         new_degree_node->previous ← old_degree_node->previous
61:         new_degree_node->previous->next ← new_degree_node
62:         old_degree_node->next ← NULL
63:         old_degree_node->previous ← NULL
64:     end if
65: end procedure
```

```
66: procedure DECREMENT(v_i, C)
67:     vertex_node ← VertexNodes[i]
68:     old_degree_node ← vertex_node->degree_node
69:     old_deg ← old_degree_node->domination_degree
70:     new_deg ← old_deg+1
71:     new_degree_node ← DegreeNodes[new_deg]
72:     old_count ← old_degree_node->count
73:     new_count ← new_degree_node->count
74:     if new_count = 0 then
75:         new_degree_node->next ← old_degree_node
76:         new_degree_node->previous ← old_degree_node->previous
77:         old_degree_node->previous->next ← new_degree_node
78:         old_degree_node->previous ← new_degree_node
79:     end if
80:     Decrement old_degree_node->count
81:     if vertex_node->is_dominated = False then
82:         {If this vertex is undominated,}
83:         {remove it from the undominated list of the old degree node}
84:         SPLICEOUT(vertex_node)
85:         Decrement old_degree_node->undominated_count
86:     end if
87:     vertex_node->degree_node ← new_degree_node
88:     if vertex_node->is_dominated = False then
89:         {If this vertex is undominated,}
90:         {add it to the undominated list of the new degree node}
91:         SPLICEIN(vertex_node)
92:         Increment new_degree_node->undominated_count
93:     end if
94:     new_degree_node->count ← new_degree_node->count +1
95:     if old_count = 0 then
96:         new_degree_node->next ← old_degree_node->next
97:         new_degree_node->next->previous ← new_degree_node
98:         old_degree_node->next ← NULL
99:         old_degree_node->previous ← NULL
100:     end if
101: end procedure
```

```
102: procedure GetMinUndominated( )
103:     degree_node ← Sentinel->next
104:     while degree_node ≠ Sentinel do
105:         if degree_node->undominated_count > 0 then
106:             return degree_node->undominated_list_sentinel->next->index
107:         end if
108:         degree_node ← degree_node->next
109:     end whilereturn −1
110: end procedure

111: procedure GetMaxUndominated( )
112:     degree_node ← Sentinel->previous
113:     while degree_node ≠ Sentinel do
114:         if degree_node->undominated_count > 0 then
115:             return degree_node->undominated_list_sentinel->next->index
116:         end if
117:         degree_node ← degree_node->previous
118:     end whilereturn −1
119: end procedure
```

3.3.3 Implementation: Algorithm 3.5

Algorithm 3.5 gives pseudocode for an implementation of the Framework using the domination degree-based Bounding Strategy 3.3 as the bounding condition and using candidate degrees to choose vertices to dominate. When a vertex v is chosen to dominate, the neighbours of v are ranked by domination degree using a linear time bucket sort. An extra stopping condition, based on Lemma 3.4, which ends the branch of recursion if any vertex has candidate degree zero, has also been included.

The candidate degree priority queue structure used to track candidate degrees allows fast selection of a vertex with either minimum or maximum candidate degree to dominate. Intuitively, choosing vertices with minimum candidate degree to dominate will help to prevent the conditions of Lemma 3.4 from being met and therefore prevent the recursive branch from being terminated. However, it may be desirable to make 'bad' choices for the vertex v to dominate, since hastening the termination of the branch may improve the computation speed. As a result, it is not immediately clear whether a minimum or maximum candidate degree vertex should be chosen as the next vertex v to dominate. The pseudocode in Algorithm 3.5 calls a function ChooseNextVertex on line 53. Two different implementations of ChooseNextVertex were programmed: one which chooses a vertex with minimum candidate degree and one which chooses a vertex with maximum candidate degree. The experimental results in Section 4.4 contain data for both implementations.

Similarly, once a vertex v is chosen as the next vertex to dominate, the candidate neighbours of v are added to the set in either ascending or descending order of domination degree. Using neighbours with high domination degree first is a logical heuristic, since doing so will maximize the number of additional vertices which are dominated. However, as with the vertex selection rule, the pseudocode is not specific about the particular ordering used, and the resolution is left to the experimental results in Section 4.4.

The domination degree multiset (Section 3.3.1) and candidate degree priority queue (Sec-

tion 3.3.2) data structures are used in Algorithm 3.5 and identified by DDMS and CDPQ, respectively (using object-oriented notation to refer to their operations). The pseudocode in Algorithm 3.5 is an implementation of Framework 3.1, but for clarity is split into three procedures: The FindDominatingSet procedure, which implements most of the framework's structure, and two helper functions AddVertex (for recursing on the case where a particular vertex is added to the set) and RestoreCandidate (for adding a vertex back to the candidate set before recursion returns).

Algorithm 3.5 Backtracking Algorithm using Bounding Strategy 3.3

```
 1: procedure AddVertex(G, P, C, B, desired_size, F, j)
 2:     if v_j ∉ C then
 3:         return
 4:     end if
 5:     Push j onto F
 6:     Remove v_j from C
 7:     DDMS.RemoveCandidate(v_j)
 8:     force_stop ← false
 9:     for each neighbour v_k ∈ N[v_j] do
10:         if v_k ∉ N[P] then
11:             CDPQ.Dominate(v_k)
12:         end if
13:         {Decrement the candidate degree of v_k}
14:         CDPQ.Decrement(v_k)
15:         if CDPQ.CandidateDegree(v_k) = 0 then
16:             force_stop ← true
17:         end if
18:         {If v_j is undominated, decrement the domination degree of v_k}
19:         if v_j ∉ N[P] then
20:             DDMS.Decrement(v_k)
21:         end if
22:     end for
23:     Add v_j to P
24:     FindDominatingSet(G, P, C, B, desired_size)
25:     Remove v_j from P
26:     return force_stop
27: end procedure
```

```
28: procedure RestoreCandidate(G, P, C, B, desired_size, j)
29:     Add v_j to C
30:     DDMS.AddCandidate(v_j)
31:     for each neighbour v_k ∈ N[v_j] do
32:         if v_k ∉ N[P] then
33:             CDPQ.Undominate(v_k)
34:         end if
35:         {Decrement the candidate degree of v_k}
36:         CDPQ.Increment(v_k)
37:         {If v_j is now undominated, increment the domination degree of v_k}
38:         if v_j ∉ N[P] then
39:             DDMS.Increment(v_k)
40:         end if
41:     end for
42: end procedure

43: procedure FindDominatingSet(G, P, C, B, desired_size)
44:     n ← |V(G)|
45:     if |N[P]| = n then
46:         if |P| < |B| then
47:             Overwrite B with a copy of P
48:         end if
49:         return
50:     end if
51:     k ← MinVerticesNeeded(G, P, C)
52:     if k ≥ |B| or k > desired_size then return
53:     v ← ChooseNextVertex(G, P, C)
54:     NeighbourList ← N[v]
55:     Sort NeighbourList by domination degree
56:     F ← Empty stack
57:     for each vertex v_j in NeighbourList do
58:         if v_j ∈ C then
59:             Remove v_j from C
60:             force_stop ←FindDominatingSet(G, P ∪ {u}, C, B, desired_size)
61:             Push j onto F
62:             if force_stop = true then
63:                 Break
64:             end if
65:         end if
66:     end for
67:     while F is non-empty do
68:         j ← Pop(F)
69:         RestoreCandidate(G, P, C, B, desired_size, j)
70:     end while
71: end procedure
```

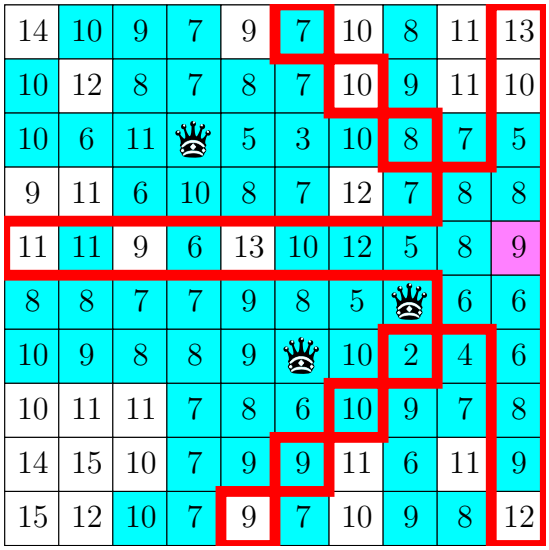


Figure 3.2: A 10×10 board with three queens.

3.4 Max Dominator Degree Algorithms

A more advanced bounding condition was devised by studying the structure of dominating sets of queen graphs and experimenting with different configurations of queens using an interactive program. The condition itself is applicable to general graphs, but has a particular advantage for queen graphs. Consider the configuration of queens on the 10×10 board shown in Figure 3.2. The highlighted pink square is undominated. Including the pink square, the board contains 29 undominated squares. Since the board contains two squares with domination degree 15, Bounding Strategy 3.3 implies that at least two queens are needed to complete the dominating set. However, the neighbourhood of the pink square (outlined in red) contains squares with domination degree at most 13, and at least one neighbour of the pink square must be added to the dominating set to dominate the pink square. Therefore, a vertex of domination degree at most 13 must be added to the dominating set, and consequently, it is not possible to cover all remaining undominated vertices with only two additional queens.

Let P be a partial dominating set and let C be a a set of candidate vertices for augmenting

P. For an undominated vertex $v \in V(G) - N[P]$, define the *max dominator degree* of v with respect to P and C, denoted by $\mathrm{MDD}_{P,C}(v)$, to be the maximum domination degree of a candidate vertex in the closed neighbourhood of v. If an undominated vertex v has a max dominator degree k, then for any dominating set D such that $P \subseteq D \subseteq P \cup C$, there must exist a vertex $u \in C$ which dominates v and has $\mathrm{DD}_P(u) \leq k$. The vertex u may also dominate other vertices whose max dominator degree is at most k.

Bounding Strategy 3.6 relates the max dominator degree of each vertex with respect to P and C to the structure of a dominating set produced from P.

Bounding Strategy 3.6. *Let C be a set of candidate vertices for augmenting a partial dominating set P. Let k be the minimum over all undominated vertices v of $\mathrm{MDD}_{P,C}(v)$.*

Then any dominating set D such that $P \subseteq D \subseteq P \cup C$ must contain at least one vertex $u \in C$ such that $\mathrm{DD}_P(u) \leq k$.

Proof. Let v be a vertex undominated by P such that $\mathrm{MDD}_{P,C}(u) = k$, and let D be a dominating set such that $P \subseteq D \subseteq P \cup C$. Let $U = \{u \in D : v \in N[u]\}$ be the set of vertices in D which dominate v. Since v is not dominated by any vertex of P, $U \subseteq C$, and since v must be dominated by at least one vertex, $|U| \geq 1$.

Since $\mathrm{MDD}_{P,C}(v)$ is defined to be the minimum of $\mathrm{DD}_P(u)$ for every candidate neighbour u of v, each vertex in $u \in U$ must have $\mathrm{DD}_P(u) \geq \mathrm{MDD}_{P,C}(v) = k$. Therefore D must contain at least one vertex with domination degree at most k. □

Bounding Strategy 3.6 can be applied to yield a bound on the size of a dominating set produced from P with candidate set C using the observation that in the best case, a vertex with domination degree k will be added to the dominating set and will, in the worst case, cover the k vertices with the smallest max dominator degrees. If the undominated vertices (in the set $V(G) - N[P]$) are ranked by their max dominator degrees in a sequence

$$v_1, v_2, \ldots, v_k$$

where $\text{MDD}_{P,C}(v_i) \leq \text{MDD}_{P,C}(v_{i+1})$ for $1 \leq i \leq k-1$, then a lower bound on the number of vertices needed to complete P into a dominating set can be computed with the pseudocode given in Algorithm 3.6.

Algorithm 3.6 MDD-based bound on the number of vertices needed to complete a dominating set.

```
1: procedure MinVerticesNeeded(G, P, C)
2:     v_1, v_2, ..., v_k ← Vertices in V(G) − N[P], ranked by max dominator degree.
3:     i ← 1
4:     count ← 0
5:     while i ≤ k do
6:         count ← count + 1
7:         i ← i + MDD_{P,C}(v_i)
8:     end while
9:     return count.
10: end procedure
```

The new MDD-based bounding condition is used to as the basis of another implementation of Framework 3.1, using the domination degree-based implementation of Algorithm 3.5 as a starting point. Since the MDD bounding condition requires a ranking of candidate vertices by max dominator degree, a data structure was developed to maintain such a ranking between different levels of recursion. Section 3.4.1 describes the new data structure. Pseudocode for the finished backtracking algorithm is given as Algorithm 3.7 in Section 3.4.2.

3.4.1 MDD Ranking Data Structure

Since the recursive structure of the backtracking algorithm ensures that vertices are added to and removed from the dominating set according to a last-in-first-out ordering, it is possible to use a stack to save some of the changes made to parameters, including max dominator degree, at each recursive step. Recursion allows such changes to be saved and restored easily, since, in most algorithms, it is sufficient to make a copy of the relevant information before recursing, then restore the information from the copy when recursion returns.

As discussed in Section 3.3.1 in the context of the domination degree multiset structure, there are two competing options for computing parameters like MDD that may change between levels of recursion. Recomputing the entire set of MDD values at each step is one option, and has the benefit of relatively low overhead, but may perform unnecessary work (since the MDD value of some vertices may not have changed since the last recomputation). Another option is to maintain the set of MDD values in a data structure that allows selective updates, at the cost of potentially higher overhead. This section describes a practical data structure to reduce MDD recomputation and allow efficient updates of MDD values as the partial dominating set P changes. For some classes of graphs, particularly those in which the maximum distance between any two vertices is small, this data structure will likely be slower than a complete recomputation, since the number of vertices whose MDD changes at each step may be large and the added overhead of the data structure is significant compared to a simple loop in such cases. For example, any two vertices $v_{i,j}$ and $v_{k,\ell}$ in a queen graph are connected by a path of length at most 2.

In Algorithm 3.7, MDD values for vertices can change under the following four circumstances. Note that the conditions below assume that vertices are removed from the dominating set in a reverse order to that in which they were added.

1. A vertex u is added to the partial dominating set P, all of u's neighbours are marked as dominated, and the MDD of vertices at distance up to 3 from u may change.

2. A vertex u is removed from the partial dominating set, the MDD values of vertices at distance up to 3 from u revert to their values before u was added to the dominating set.

3. After a vertex u is removed from the partial dominating set, it is removed from the candidate set C to prevent it from being considered for the dominating set by any future iterations of the algorithm. In this case, if u is a max dominator of any vertex

v, the MDD of v may change since u is no longer viable as a max dominator.

4. Before recursion returns, all vertices which have been removed from C during that call to FindDominatingSet are added back to C (with the most recently removed vertices added back first). As a result, for all vertices u which were removed from C, the MDD values of the neighbours of u are restored to their values before u was removed from C.

By using a stack to save the modified values in cases 1 and 3, cases 2 and 4 can be implemented by popping the modified values from the stack and restoring them. The MDDRanking data structure tracks the MDD of every uncovered vertex $v \in V(G) - N[S]$, and contains the following operations.

Operation	**Specification**
Init(G)	Initialize for the graph G, assuming that $P = \emptyset$ and $C = V(G)$.
AddDominator(v)	Update the MDD of each vertex to accommodate v being added to P.
RemoveDominator(v)	Inverse of AddDominator: Revert the MDD of each vertex to its state before v was added to P.
ExcludeDominator(v)	Update the MDD of each vertex to accommodate v being removed from C and P.
UnexcludeDominator(v)	Inverse of ExcludeDominator: Revert the MDD of each vertex to its state before v was removed from C and P.
GetMDD(v)	Return the MDD of vertex v.
MinVerticesNeeded(S, F)	Return the minimum number of vertices in C which must be added to P to form a dominating set.

The MDDRanking structure uses four data structures internally:

- An array `MDD` stores the current MDD of each vertex v_i.

- An array `MDD_counts` tracks the number of vertices with each MDD (that is, element `MDD_counts`$[i]$ contains the number of vertices with MDD i).
- A stack `ST` whose elements are sets which contain tuples of the form $(v, \texttt{old\_MDD})$. Each set on the stack contains the set of vertices (and their old MDD values) modified by a single call to AddDominator or ExcludeDominator.

The Init operation sets the initial MDD of each vertex v to

$$\max_{u \in N[v]} (\deg(u) + 1) .$$

The Init operation requires $\Theta(|V(G)| + |E(G)|)$ operations in the worst case, but this is of little significance to the overall performance of the algorithm since initialization is only done once. The GetMDD operation can be implemented with only a table lookup, since the MDD of each vertex is stored. The MinVerticesNeeded operation is equivalent to Algorithm 3.6, except that sorting the vertices by MDD (line 2 of Algorithm 3.6) is unnecessary since the `MDD_counts` array in the data structure maintains the ranking persistently.

3.4.2 Implementation: Algorithm 3.7

As with the domination degree-based approach in Algorithm 3.5, the extra information which is maintained to evaluate the MDD-based bounding condition allows refinements to the vertex selection rule and the ranking of neighbours during the backtracking algorithm. Algorithm 3.7 contains pseudocode for a backtracking algorithm based on Framework 3.1 which uses Bounding Strategy 3.6 as the bounding condition.

Algorithm 3.7 is similar to Algorithm 3.5, except that the DDMS structure (which tracks domination degrees) has been removed and a MDDRanking structure is used instead, in keeping with the new bounding condition. The algorithm still tracks the candidate degree of every vertex with a candidate degree priority queue structure (see Section 3.3.2) called

CDPQ.

Since a ranking of vertices by both MDD (via the MDDRanking) and candidate degree (via the CDPQ structure) is available, there are more options for the vertex selection rule. As with Algorithm 3.5, the selection of an undominated vertex is delegated to a function ChooseNextVertex, which is not specified. In the experimental testing in Section 4.5, four different vertex selection rules were tested: minimum candidate degree, maximum candidate degree, minimum MDD and maximum MDD.

Although a ranking of vertices by domination degree is not maintained, the domination degree of each vertex is still tracked, allowing the neighbour ranking step to continue to use domination degree as in Algorithm 3.5.

Algorithm 3.7 Backtracking Algorithm using Bounding Strategy 3.6

```
1: procedure AddVertex(G, P, C, B, desired_size, F, j)
2:     if v_j ∉ C then
3:         return
4:     end if
5:     Push j onto F
6:     Remove v_j from C
7:     force_stop ← false
8:     for each neighbour v_k ∈ N[v_j] do
9:         if v_k ∉ N[P] then
10:            CDPQ.Dominate(v_k)
11:        end if
12:        {Decrement the candidate degree of v_k}
13:        CDPQ.Decrement(v_k)
14:        if CDPQ.CandidiateDegree(v_k) = 0 then
15:            force_stop ← true
16:        end if
17:    end for
18:    Add v_j to P
19:    MDDRanking.AddDominator(v_j)
20:    FindDominatingSet(G, P, C, B, desired_size)
21:    MDDRanking.RemoveDominator(v_j)
22:    Remove v_j from P
23:    MDDRanking.ExcludeDominator(v_j)
24:    return force_stop
25: end procedure

26: procedure RestoreCandidate(G, P, C, B, desired_size, j)
27:    MDDRanking.UnexcludeDominator(v_j)
28:    Add v_j to C
29:    for each neighbour v_k ∈ N[v_j] do
30:        if v_k ∉ N[P] then
31:            CDPQ.Undominate(v_k)
32:        end if
33:        {Increment the candidate degree of v_k}
34:        CDPQ.Increment(v_k)
35:    end for
36: end procedure
```

```
37: procedure FindDominatingSet(G, P, C, B, desired_size)
38:     n ← |V(G)|
39:     if |N[P]| = n then
40:         if |P| < |B| then
41:             Overwrite B with a copy of P
42:         end if
43:         return
44:     end if
45:     k ← MinVerticesNeeded(G, P, C)
46:     if k ≥ |B| or k > desired_size then return
47:     v ← ChooseNextVertex(G, P, C)
48:     NeighbourList ← N[v]
49:     Sort NeighbourList by domination degree
50:     F ← Empty stack
51:     for each vertex v_j in NeighbourList do
52:         if v_j ∈ C then
53:             Remove v_j from C
54:             force_stop ←FindDominatingSet(G, P ∪ {u}, C, B, desired_size)
55:             Push j onto F
56:             if force_stop = true then
57:                 Break
58:             end if
59:         end if
60:     end for
61:     while F is non-empty do
62:         j ← Pop(F)
63:         RestoreCandidate(G, P, C, B, desired_size, j)
64:     end while
65: end procedure
```

Chapter 4

Experimental Evaluation of Domination Algorithms

This chapter contains the results of several large-scale experimental comparisons of the different implementations of the backtracking algorithms based on Framework 3.1. The overarching goal of the experiments was to gather data that would lead to a small number of high performance solvers for the optimization problem of finding a minimum dominating set of an arbitrary graph. As a control, and to verify the results of the new algorithms, the dominating set solver in the SageMath suite [1] was also profiled. To measure the performance of each implementation and its variants, it was necessary to profile the performance of each variant on a wide selection of graphs, sampled from a variety of graph families.

Following the discussion in Chapter 1 about the different ways to characterize 'performance' of an algorithm, and how analytical asymptotic measurements (even of expected-case running time) can sharply depart from realistic cases in exponential-time searches, the experiments in this chapter were designed to measure practical performance, with several precautions taken to ensure that the results would speak to the general performance of the various algorithms tested.

A total of 51 different variants on Framework 3.1 were each run on a total of 550 input graphs for up to 2 hours (7200 seconds) each. The huge volume of data collected from these experiments forms the core of the empirical results of this thesis. The analysis in this chapter contains comparisons of the variants within their broad categories (corresponding to the three primary bounding methods described in Chapter 3), an overall comparison of all variants based on running time and call tree size, and a comparison of the variants with the best running time for each graph against the SageMath dominating set solver [1] in Section 4.7.

Since the goal of these experiments is to produce a high performance general solver for the dominating set optimization problem (with no initial conditions such as a known upper bound on $\gamma(G)$), the experimental data in this chapter focuses exclusively on that problem. Other uses of the dominating set algorithms, such as exhaustively generating all dominating sets of a given size, or finding a smallest dominating set within a given range of sizes (for example, using a known upper bound as the starting point of the computation), are not covered by this data. In the context of Framework 3.1, this entails setting the upper bound value of `desired_size` to $|V(G)|$ for every input graph.

Section 4.1 describes the base input dataset used for the experiments. Section 4.2 details the experiment setup, including measures taken to mitigate the effect of 'luck' on an algorithm's performance by randomizing inputs. The variants of Framework 3.1 presented in Chapter 3 are grouped into three categories: variants using a fixed vertex ordering (Section 3.2), variants using domination degree for bounding and to choose vertices (Section 3.3) and variants which use max dominator degree for bounding and to choose vertices (Section 3.4). Accordingly, detailed results and analysis for this series of experiments are presented separately for each category. Results for three variants using fixed vertex ordering (based on Algorithm 3.2) are presented in Section 4.3, results for 16 variants using domination degree (based on Algorithm 3.5) are presented in Section 4.4 and results for 32 variants using

max dominator degree (based on Algorithm 3.7) are presented in Section 4.5. Section 4.6 contains a comparison of the best variants within each category for each graph, and Section 4.7 compares the best Framework 3.1-based algorithm against the SageMath solver. Finally, Section 4.8 uses the experimental data to choose several high performance representative algorithms.

4.1 Input Graph Dataset

The set of input graphs for these experiments consists of several collections of graphs drawn from various graph families. The collections were chosen to meet the following criteria.

- Each collection is taken from an infinite family of graphs for which there are open questions regarding the domination number.
- The domination number of graphs in the collection can be found in a reasonable amount of time by at least one of the tested algorithms, but the graphs are 'hard' enough that running time differences are significant.
- Published data is available on the domination numbers of known cases, to verify the results of the algorithms.
- The different collections span a range of edge densities, from relatively sparse to relatively dense graphs.
- All of the graphs are connected, since a minimum dominating set of a disconnected graph can be found by combining minimum dominating sets of each component.

To that end, the input dataset includes a selection of Queen graphs, Kneser graphs, Knight graphs, Covering Code graphs, Triangle Grid graphs, Hex Rook graphs and Cartesian

products of cycles, as described in Chapter 2. Table 4.1 lists the complete set of graphs chosen, along with some of their graph-theoretic parameters.

Table 4.1: Parameters of the graphs used as inputs for the optimization experiment.

Graph	n	m	Δ	γ
Cartesian Products of Cycles				
$C_8 \square C_8$	64	128	4	16
$C_9 \square C_9$	81	162	4	18
$C_{10} \square C_{10}$	100	200	4	20
$C_{11} \square C_{11}$	121	242	4	27
$C_{12} \square C_{12}$	144	288	4	32
$C_{13} \square C_{13}$	169	338	4	38
$C_{14} \square C_{14}$	196	392	4	42
$C_{15} \square C_{15}$	225	450	4	45
Covering Code Graphs				
$\text{Code}_1(2,6)$	64	192	6	12
$\text{Code}_2(2,6)$	64	672	21	4
$\text{Code}_3(2,6)$	64	1312	41	2
$\text{Code}_1(2,7)$	128	448	7	16
$\text{Code}_2(2,7)$	128	1792	28	7
$\text{Code}_3(2,7)$	128	4032	63	2
$\text{Code}_1(2,8)$	256	1024	8	32
$\text{Code}_3(2,8)$	256	11776	92	4
$\text{Code}_2(3,5)$	243	6075	50	8
Hex Rook Graphs				
HR (10)	55	495	18	5
HR (11)	66	660	20	5
HR (12)	78	858	22	6
HR (13)	91	1092	24	6
HR (14)	105	1365	26	7
HR (15)	120	1680	28	7
HR (16)	136	2040	30	8
HR (17)	153	2448	32	8
HR (18)	171	2907	34	9
HR (19)	190	3420	36	9
HR (20)	210	3990	38	9

Graph	n	m	Δ	γ
Kneser Graphs				
Kneser $(8,3)$	56	280	10	7
Kneser $(9,3)$	56	280	10	7
Kneser $(9,4)$	126	315	5	26
Kneser $(10,3)$	56	280	10	7
Kneser $(11,3)$	56	280	10	7
Knight Graphs				
Knight (4)	16	24	4	4
Knight (5)	25	48	8	5
Knight (6)	36	80	8	8
Knight (7)	49	120	8	10
Knight (8)	64	168	8	12
Knight (9)	81	224	8	14
Knight (10)	100	288	8	16
Knight (11)	121	360	8	21
Queen Graphs				
Queen (10)	100	1470	35	5
Queen (11)	121	1980	40	5
Queen (12)	144	2596	43	6
Queen (13)	169	3328	48	7
Queen (14)	196	4186	51	8
Queen (15)	225	5180	56	9
Triangle Grid Graphs				
TG (11)	66	165	6	13
TG (12)	78	198	6	15
TG (13)	91	234	6	17
TG (14)	105	273	6	19
TG (15)	120	315	6	21
TG (16)	136	360	6	24
TG (17)	153	408	6	27
TG (18)	171	459	6	30
TG (19)	190	513	6	33
TG (20)	210	570	6	36

4.2 Methodology

All of the experiments detailed in Sections 4.3 - 4.5 were run and timed on the same machine, a four-core Intel Core i7-3770 running at 3.4 Ghz. The experiments were run over the course

of several months, during which time three experiments would be run at a time (freeing the remaining core for background operating system tasks). Other than the experiments, no other jobs were running on the machine during the entire span of the experiment. The system was equipped with 16GB of memory, and each tested executable was allocated 4gb of memory. Executables attempting to exceed this limit would be terminated, although no executable did so during the course of the experiments. A separate process was started for each input graph, instead of having one process read a succession of graphs, and each process was allowed two hours (7200 seconds) of real execution time, including all input and output time. The running times themselves were computed within the executable using the POSIX real-time library (historically called `librt`), with timing beginning immediately before the search algorithm begins (after the input graph had been read but before any preprocessing specific to the algorithm) and ending immediately after the search algorithm ends.

Sections 4.3 - 4.5 contain the results of experiments comparing several variants of each algorithm. In cases where an algorithm did not finish within the allocated 2 hours of running time, the search was terminated and the trial was labelled as a failure (such cases are documented in the results in the following sections). No other data was collected for failed trials. In cases where the algorithm finished within the allocated time, the total running time (excluding input and output time) was logged. The output of each successful trial was verified, and for all cases, was determined to be a valid dominating set of the correct size. The total number of calls to the FindDominatingSet procedure was also logged. Since all of the algorithms are built on Framework 3.1, the number of calls to FindDominatingSet can be used as a time-agnostic comparison of each algorithm's ability to reduce the search tree size.

4.2.1 Mitigating the impact of 'luck'

Internally, each algorithm was implemented using an adjacency list structure to store each graph. The initial numbering of vertices in the input can affect the performance of the algorithm, even in cases where the algorithm renumbers some vertices as a pre-processing step, since when the renumbering is based on a property (such as degree) where ties may occur, more than one possible numbering may exist. For a problem such as exhaustively generating all dominating sets (or all sets of a given size), the initial numbering of vertices would not generally affect the total time of the algorithm, since every dominating set must be generated in all cases. For these experiments, however, the tested problem was of finding a single example of a minimum dominating set. Once a minimum dominating set is found, every algorithm must assert that no smaller sets exist before terminating (which leads to a similar insensitivity to initial numbering as the exhaustive generation case), but, depending on the initial ordering of vertices, the time needed to converge on a minimum dominating set may differ. For some graphs, once a minimum dominating set is found, very little time is needed to assert that no smaller sets exist compared to the time required to find the minimum dominating set. On a particular input, an algorithm's performance may be artificially inflated by being 'lucky' and finding a small set quickly due to the initial ordering of vertices.

To mitigate this sensitivity to initial conditions, each algorithm was tested multiple times on different numberings of the same input graph. Ten permuted variants of each graph from the input collection were produced by applying a random permutation to the numbering of vertices of the graph, sampled uniformly at random from the set of all $n!$ permutations of the n vertices of the graph using a Knuth shuffle algorithm [46]. The resulting set of 550 permuted graphs was then used as the input for the experiments. The original, unpermuted graph was not used as a direct input for any of the tested graphs, since the generation mechanism for each graph may produce a numbering that favours a particular implementation.

Since the permuted orderings were randomly generated, the distribution of running times of the same algorithm over all permuted versions of the same graph should produce a reliable measurement of the performance of that algorithm, with the influence of 'luck' minimized. Various methods were used to collapse the set of running times over all tested permutations into a single metric.

- The *minimum* running time of a particular algorithm over all permuted versions of one graph provides a benchmark for 'best case' running time, which is highly susceptible to the 'luck' issue described above. In general, an algorithm that displays consistently good minimum running times but less impressive average or maximum running times is likely very sensitive to the initial ordering of vertices, which means that future research may be necessary into how best to order vertices before running the algorithm.

- The *maximum* running time of an algorithm over all permuted copies of one graph provides a benchmark for 'worst case' running time. This can be used to determine performance under the least 'lucky' conditions for initial numbering. In the context of finding an algorithm with good general performance, a low maximum running time might be the most valuable measurement, since it provides some assurance that the algorithm will complete in a predictable amount of time.

- The *average* running time of an algorithm over all permuted copies of one graph might be the most reliable metric for comparison against other algorithms. However, the average may be unduly affected by outliers, for better or worse. In cases where one or more permutations of the graph resulted in a failed trial due to the algorithm not finishing within the alloted time, the average is essentially invalid.

- Various measures of *spread* can be used to determine the degree of uncertainty of the collected data for a particular algorithm on a particular graph. For example, if all 10 permuted copies of the graph have running times which are tightly clustered, the

algorithm might be expected to have relatively predictable performance on any random permutation of the graph, and therefore lack sensitivity to the initial numbering.

4.3 Fixed-Ordering Implementations

Algorithm 3.2 in Section 3.2 is an implementation of Framework 3.1 which uses static bounding conditions and chooses vertices to dominate according only to their current domination status (that is, dominated or undominated) and their index. The other variants (Algorithms 3.5 and 3.7) choose vertices to dominate based on other conditions which are derived from the structure of the working dominating set (such as domination degree). Algorithm 3.2 uses very simple logic to select the next vertex v to dominate and to iterate over the neighbours of v to recurse on dominators, and therefore, unlike the other variants, there is no need to test different combinations of heuristics, the algorithm is sensitive to the numbering of vertices in the input.

To investigate the impact of vertex numbering on the performance of the algorithm, three different variants of the algorithm were created, each with a preprocessing stage which renumbered the vertices of the input graph. The different numberings used were

- Ascending order of degree;
- Descending order of degree; and
- The order produced by a breadth-first search (BFS) traversal starting at vertex 0 in the initial input graph, with the neighbours of each vertex visited by the traversal in numerical order. The reason that BFS was chosen instead of a different traversal algorithm (such as depth-first search) was that the traversal tree produced by BFS is guaranteed to contain paths with the smallest number of edges between the starting vertex and every other vertex, thereby preserving a sense of "locality" in the numbering.

Tables 4.2 - 4.8 show the minimum, maximum and average running times of each variant, along with the number of did-not-finish (failed trial) cases, over all 10 permutations of each input graph. For each of the three metrics (minimum, maximum and average), the minimum value in each row is shaded in gray. Broadly, the results show that the BFS ordering is generally superior on the tested graphs, except classes with relatively sparse graphs, such as the Knight and Triangular Grid graphs, where minimum degree ordering had better performance. Maximum degree ordering had the best performance on a handful of cases, notably on the hex rook graphs HR (12) and HR (16) where the other two implementations had high failure rates. The minimum and maximum degree orderings have a natural disadvantage on graphs where the degree was relatively uniform (or regular), reordering by degree has little or no effect on those graphs.

The data in Tables 4.2 - 4.8 is also instructive on the usefulness of the different measurements of performance. The 'minimum' time, in particular, is not representative of the performance of the algorithm in many cases, particularly when several trials did not finish. The average time, which is automatically rendered invalid by a single DNF trial, is better at capturing the overall performance. There were very few cases where the algorithm with the best average time did not also have the best maximum time. To predict the usefulness of the algorithm as a general purpose solver, where the user's impression of performance will not be the mean of several trials, the maximum time seems to be the best statistic, since, as mentioned in the previous section, it provides the best insight into the 'worst case' performance of the algorithm.

Based on this logic, the maximum time statistic was used as the primary metric for the far more involved analysis of the remaining two variants and the overall comparison of the different variants in Section 4.6

Table 4.2: Running times (in seconds) of the three fixed-ordering variants on covering code graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	**n**	**m**	**Min. Degree Ordering**				**Max. Degree Ordering**				**BFS Ordering**			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
$\text{Code}_1\,(2,6)$	64	192	1.423	1.795	2.196	0	1.7	2.027	2.584	0	0.265	0.265	0.267	0
$\text{Code}_2\,(2,6)$	64	672	0.001	0.001	0.002	0	0.001	0.001	0.001	0	0.001	0.001	0.001	0
$\text{Code}_3\,(2,6)$	64	1312	0	–	–	5	0	–	–	1	0	–	–	1
$\text{Code}_1\,(2,7)$	128	448	1.41	7.119	21.39	0	1.947	11.85	36.07	0	9.269	10.15	11.04	0
$\text{Code}_2\,(2,7)$	128	1792	15.69	17.66	19.24	0	15.55	17.57	19.45	0	5.017	5.094	5.16	0
$\text{Code}_3\,(2,7)$	128	4032	0.001	0.001	0.001	0	0.001	0.001	0.001	0	0.001	0.001	0.001	0
$\text{Code}_3\,(2,8)$	256	11776	0.163	0.185	0.208	0	0.164	–	–	1	0.124	0.15	0.155	0
$\text{Code}_2\,(3,5)$	243	6075	–	–	–	10	–	–	–	10	–	–	–	10

Table 4.3: Running times (in seconds) of the three fixed-ordering variants on hex rook graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	**n**	**m**	**Min. Degree Ordering**				**Max. Degree Ordering**				**BFS Ordering**			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
HR (10)	55	495	0.008	0.009	0.011	0	0.009	0.009	0.01	0	0.007	–	–	4
HR (11)	66	660	0.015	0.02	0.038	0	0.015	0.021	0.051	0	0.011	0.016	0.028	0
HR (12)	78	858	0.263	–	–	1	0.271	0.287	0.306	0	0.196	–	–	7
HR (13)	91	1092	0.441	0.519	0.617	0	0.472	0.542	0.641	0	0.234	0.319	0.463	0
HR (14)	105	1365	9.865	10.62	11.68	0	10.13	–	–	1	6.138	–	–	7
HR (15)	120	1680	18.2	19.75	21.09	0	17.04	20.18	21.97	0	7.668	12.48	15.01	0
HR (16)	136	2040	479.6	–	–	1	410.7	482.5	543.4	0	205.1	–	–	5
HR (17)	153	2448	760.3	871.7	970.5	0	799.8	893.7	962.2	0	370.5	527.8	690	0
HR (18)	171	2907	–	–	–	10	–	–	–	10	–	–	–	10
HR (19)	190	3420	–	–	–	10	–	–	–	10	–	–	–	10
HR (20)	210	3990	–	–	–	10	–	–	–	10	–	–	–	10

Table 4.4: Running times (in seconds) of the three fixed-ordering variants on Kneser graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	**n**	**m**	**Min. Degree Ordering**				**Max. Degree Ordering**				**BFS Ordering**			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
Kneser $(8,3)$	56	280	0.014	0.017	0.024	0	0.013	0.017	0.025	0	0.016	0.017	0.018	0
Kneser $(9,3)$	56	280	0.013	0.015	0.017	0	0.015	0.017	0.024	0	0.016	0.018	0.021	0
Kneser $(9,4)$	126	315	–	–	–	10	–	–	–	10	–	–	–	10
Kneser $(10,3)$	56	280	0.014	0.016	0.02	0	0.014	0.016	0.019	0	0.012	0.017	0.021	0
Kneser $(11,3)$	56	280	0.013	0.016	0.019	0	0.013	0.017	0.025	0	0.012	0.018	0.021	0

Table 4.5: Running times (in seconds) of the three fixed-ordering variants on knight graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	n	m	Min. Degree Ordering				Max. Degree Ordering				BFS Ordering			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
Knight (4)	16	24	0	0	0	0	–	–	–	10	0	0	0	0
Knight (5)	25	48	0	0	0	0	–	–	–	10	0	0	0	0
Knight (6)	36	80	0	0	0.001	0	–	–	–	10	0.001	0.002	0.003	0
Knight (7)	49	120	0.009	0.01	0.012	0	0.278	0.319	0.406	0	0.015	0.039	0.074	0
Knight (8)	64	168	0.213	0.238	0.286	0	8.13	10.26	17.36	0	0.393	1.307	3.014	0
Knight (9)	81	224	0.299	0.461	0.559	0	121.1	285.8	721.1	0	4.337	28.36	89.77	0
Knight (10)	100	288	4.535	6.058	8.263	0	1737	–	–	4	36.52	327.7	881.9	0
Knight (11)	121	360	2251	2528	3017	0	–	–	–	10	–	–	–	10

Table 4.6: Running times (in seconds) of the three fixed-ordering variants on Cartesian products of cycles, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	n	m	Min. Degree Ordering				Max. Degree Ordering				BFS Ordering			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
$C_8 \square C_8$	64	128	9.147	11.03	13.86	0	7.053	12.31	19.8	0	0.692	0.706	0.725	0
$C_9 \square C_9$	81	162	10.44	17.99	25.82	0	10.17	17.6	28.71	0	0.52	0.58	0.679	0
$C_{10} \square C_{10}$	100	200	25.97	67.12	185.2	0	22.33	85.38	180.5	0	0.622	0.669	0.71	0
$C_{11} \square C_{11}$	121	242	–	–	–	10	–	–	–	10	303.1	388.5	449.6	0
$C_{12} \square C_{12}$	144	288	–	–	–	10	–	–	–	10	–	–	–	10
$C_{13} \square C_{13}$	169	338	–	–	–	10	–	–	–	10	–	–	–	10
$C_{14} \square C_{14}$	196	392	–	–	–	10	–	–	–	10	–	–	–	10
$C_{15} \square C_{15}$	225	450	–	–	–	10	–	–	–	10	–	–	–	10

Table 4.7: Running times (in seconds) of the three fixed-ordering variants on queen graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	n	m	Min. Degree Ordering				Max. Degree Ordering				BFS Ordering			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
Queen (10)	100	1470	0.072	0.119	0.171	0	0.145	0.159	0.172	0	0.105	0.2	0.422	0
Queen (11)	121	1980	1.014	1.388	1.752	0	0.254	0.269	0.281	0	0.074	1.265	3.125	0
Queen (12)	144	2596	9.629	24.02	38.6	0	11.33	11.73	12.13	0	3.554	9.764	19.39	0
Queen (13)	169	3328	155.1	206.8	254.4	0	349.6	354.8	364.2	0	141.7	165	216.7	0
Queen (14)	196	4186	4939	5833	6693	0	–	–	–	10	4779	5719	6646	0
Queen (15)	225	5180	–	–	–	10	–	–	–	10	–	–	–	10

Table 4.8: Running times (in seconds) of the three fixed-ordering variants on triangle grid graphs, along with the number of cases (out of 10) on which each implementation did not finish (DNF).

Graph	n	m	Min. Degree Ordering				Max. Degree Ordering				BFS Ordering			
			Running Time			DNF	Running Time			DNF	Running Time			DNF
			Min.	Avg.	Max.		Min.	Avg.	Max.		Min.	Avg.	Max.	
TG(11)	66	165	0.069	0.105	0.132	0	3.02	5.56	10.52	0	0.075	0.112	0.174	0
TG(12)	78	198	0.294	0.457	0.655	0	45.71	107.9	175.4	0	0.438	1.208	2.74	0
TG(13)	91	234	2.666	4.053	6.038	0	591.4	1178	1824	0	2.103	6.201	19.44	0
TG(14)	105	273	46.36	57.37	82.84	0	–	–	–	10	10.6	45.46	133.2	0
TG(15)	120	315	130.3	184.2	249.8	0	–	–	–	10	53.84	428.3	1383	0
TG(16)	136	360	2923	4711	6797	0	–	–	–	10	974.7	–	–	2
TG(17)	153	408	–	–	–	10	–	–	–	10	–	–	–	10
TG(18)	171	459	–	–	–	10	–	–	–	10	–	–	–	10
TG(19)	190	513	–	–	–	10	–	–	–	10	–	–	–	10

4.4 Domination Degree Implementations

The variant of Framework 3.1 which uses domination degree in the bounding condition is described in Section 3.3. Pseudocode for this variant is given in Algorithm 3.5, but as mentioned in Section 3.3, there are several aspects of the algorithm for which multiple implementation options exist.

First, and most significantly, the criteria for choosing the next vertex to dominate (the ChooseNextVertex function on line 53 of Algorithm 3.5) is deliberately left undefined in Algorithm 3.5. The two options proposed in Section 3.5 for this procedure are to choose a vertex with minimum candidate degree or a vertex with maximum candidate degree.

Second, in Algorithm 3.5, once a vertex v has been chosen to dominate, the neighbours of v are sorted by domination degree (on line 55 of Algorithm 3.5). The sort order (ascending/descending) is left undefined, since it is unclear (beyond anecdotal assumptions) which order will lead to higher performance.

The third aspect is an optimization. On line 62, the current branch of recursion is terminated in cases where excluding a vertex from consideration as a dominator will prevent a dominating set from being completed. This optimization does not carry any real computation

cost (besides constant-time overhead) in Algorithm 3.5, but it is not clear how effective it is at ending branches.

Finally, the pseudocode for Algorithm 3.5 tests the bounding condition at the beginning of each call to FindDominatingSet (lines 51 - 52 of Algorithm 3.5), as specified in Framework 3.1. However, once a vertex v has been chosen for domination, each iteration of the loop over the set of neighbours of v (line 57) changes the set of candidate vertices, and therefore potentially changes outcome of the bounding conditions. This is not an issue in the fixed order algorithms (where the static bounding condition is only affected by the size of the working set, which is fixed over all iterations of the loop). In cases where, for example, the first iteration of the loop excludes a candidate vertex which upsets the bounding condition, the algorithm will recurse on all of the other neighbours of v and promptly return (since the check of the bounding conditions in the recursive call will fail). One possible optimization is to recheck the bounding condition at each iteration of the loop (that is, before line 58) and terminate the current branch of recursion immediately if the bound fails.

The four aspects above are independent, and there are two possible implementations for each aspect. To test all possibilities, sixteen implementations of Algorithm 3.5 were prepared, representing all combinations of the aspects above:

- Minimum vs. maximum candidate degree vertices chosen for domination.
- Ascending vs. descending neighbour ordering.
- The 'force-stop' (FS) optimization enabled vs. disabled.
- Bound rechecking (RC) optimization at each iteration of the loop enabled vs. disabled.

Each of the sixteen implementations was tested on the entire input dataset as specified in Section 4.2.

An overall comparison of the results of the experiments on each graph, compared with the results from other variants, can be found in Section 4.6.

4.4.1 Single Aspect Comparisons

Although the four aspects detailed in the previous section are independent in the code, they may be correlated in their behavior. For example, it may be that a combination of the force-stop optimization and the bound rechecking during the neighbour loop results in a speed improvement, but in all other cases (where the two are not used together) no improvement is observed, due to some feedback between the two aspects. Since the ultimate goal of this series of experiments is to find the "best" combination of aspects, it is important to note apparent correlations when they occur.

On the other hand, it is also useful to assess each aspect separately, to investigate the overall impact of each choice on the performance of the dominating set computation when all other aspects are fixed. Since every possible combination of aspects was tested as part of the experiment, it is possible to examine the difference in running time or call tree size between pairs of algorithms which have a different implementation of a particular aspect (such as vertex selection) but identical implementations of every other aspect.

Denote each of the 16 variants of Algorithm 3.5 by

$$A(\text{vs}, \text{no}, \text{fs}, \text{rc})$$

where vs $\in \{\text{Min. CD}, \text{Max. CD}\}$, no $\in \{\textit{ascending}, \textit{descending}\}$, fs $\in \{\text{enabled}, \text{disabled}\}$ and rc $\in \{\text{enabled}, \text{disabled}\}$. For a graph G in the input dataset of Section 4.1, let $\text{MaxTime}(G, A(\text{vs}, \text{no}, \text{fs}, \text{rc}))$ and $\text{MaxCalls}(G, A(\text{vs}, \text{no}, \text{fs}, \text{rc}))$ be the maximum running time and number of recursive calls to FindDominatingSet (respectively) across the 10 tested permutations of the graph G.

To examine the impact of a particular aspect, the ratios of running time or calls between pairs of variants which differ in exactly one aspect can be examined. For example, if $\mathcal{G}$ is the set of all graphs in the input dataset, the ratios of running time for ascending neighbour

order versus descending neighbour order are the values

$$\frac{\text{MaxTime}(G, A(\text{vs}, \text{ascending}, \text{fs}, \text{rc}))}{\text{MaxTime}(G, A(\text{vs}, \text{descending}, \text{fs}, \text{rc}))}$$

where $G \in \mathcal{G}$, vs $\in \{\text{Min. CD}, \text{Max. CD}\}$, fs $\in \{\text{enabled}, \text{disabled}\}$ and rc $\in \{\text{enabled}, \text{disabled}\}$. The ratio can similarly be produced for the call tree size.

Figures 4.1 - 4.4 show histograms of the multiset of pairwise ratios computed by the formula above for each of the four aspects detailed in the previous section. Cases where neither algorithm finished are excluded from the histogram. Cases where the algorithm in the numerator did not finish but the algorithm in the denominator did finish are treated as having a ratio of zero, and symmetrically, cases where the algorithm in the numerator finished but the algorithm in the denominator did not are treated as having a ratio of infinity (which falls into the rightmost bar of the histogram). Intuitively, for a histogram which compares "Option A vs. Option B", the left side of the histogram corresponds to cases where Option A had a smaller running time or number of calls and the right side of the histogram corresponds to cases where Option B had a smaller running time or number of calls.

Tables 4.9 and 4.10 contain a numerical summary of the data in the histograms per family of graphs. For each type of ratio and graph family, the tables show the number of zero ratios, the number of infinite ratios and the median ratio among all cases where both the numerator and denominator algorithms finished within the alloted time. The tables also contain data for the complete input set in two forms: a general aggregated median across all input graphs, and a median across all families (with each family receiving equal weight) to control for the differing number of graphs in different families (which might bias the aggregate median).

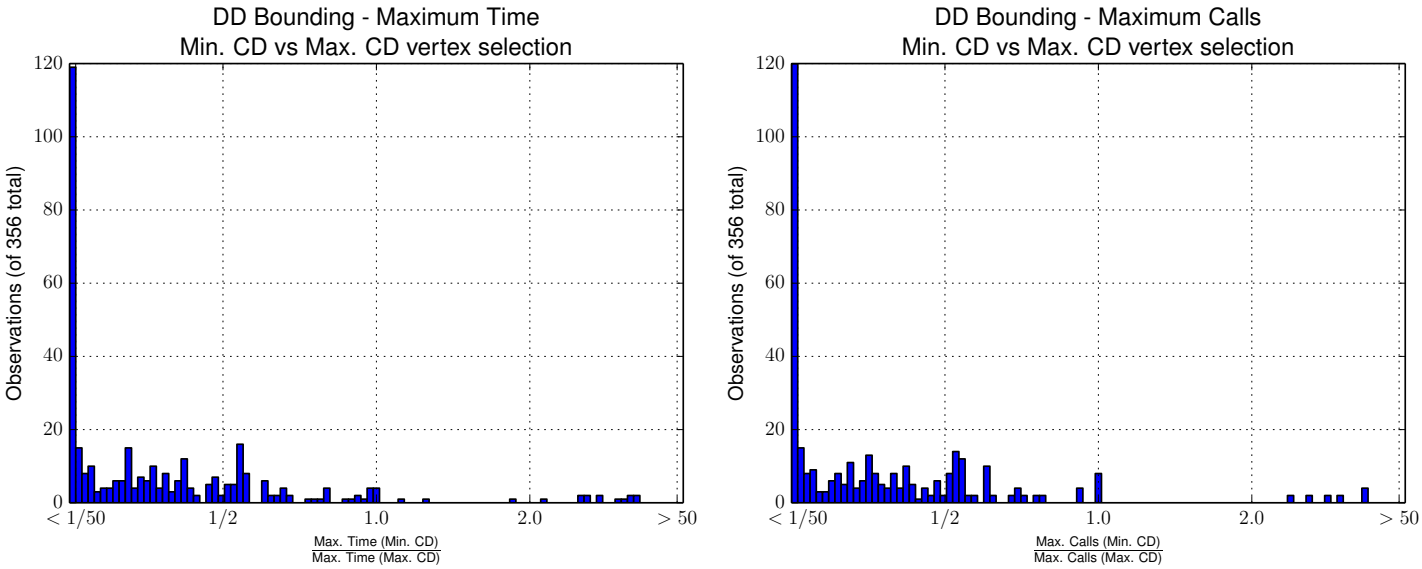
All of the histograms show relatively strong trends, and the trend for running time generally agrees with the trend for total calls. Maximum CD vertex selection has worse performance than minimum CD vertex selection in almost all cases, with an extremely high number of cases falling into the extreme right of the histogram. Descending neighbour

ordering generally has better performance than ascending neighbouring, but in the roughly 26% (100/380) of cases where ascending order has better performance, the advantage is extremely lopsided (in both time and calls). The per-family summary in Tables 4.9 and 4.10 reveals that the disparity is primarily due to the descending order variants not finishing within the alloted time on a large number of cases. Conversely, there are very few cases in which the ascending order variants do not finish. Arguably, the ascending ordering is a better choice for a general algorithm, since the experimental data suggests it is bounded to within a more limited range (even though it is often slower than descending ordering).

For both of the optimization aspects, the histograms show unimpressive running time performance. The bound rechecking optimization resulted in a reasonably consistent improvement in the number of calls, but overall had worse running time, likely because of the overhead of recomputing the bounds at each iteration of the loop. The force stop optimization had minimal impact on the number of calls. This is likely a sign that the optimization almost never terminated any branches. Overall, the histograms suggest that leaving both optimizations disabled is a better option for graphs similar to the input dataset.

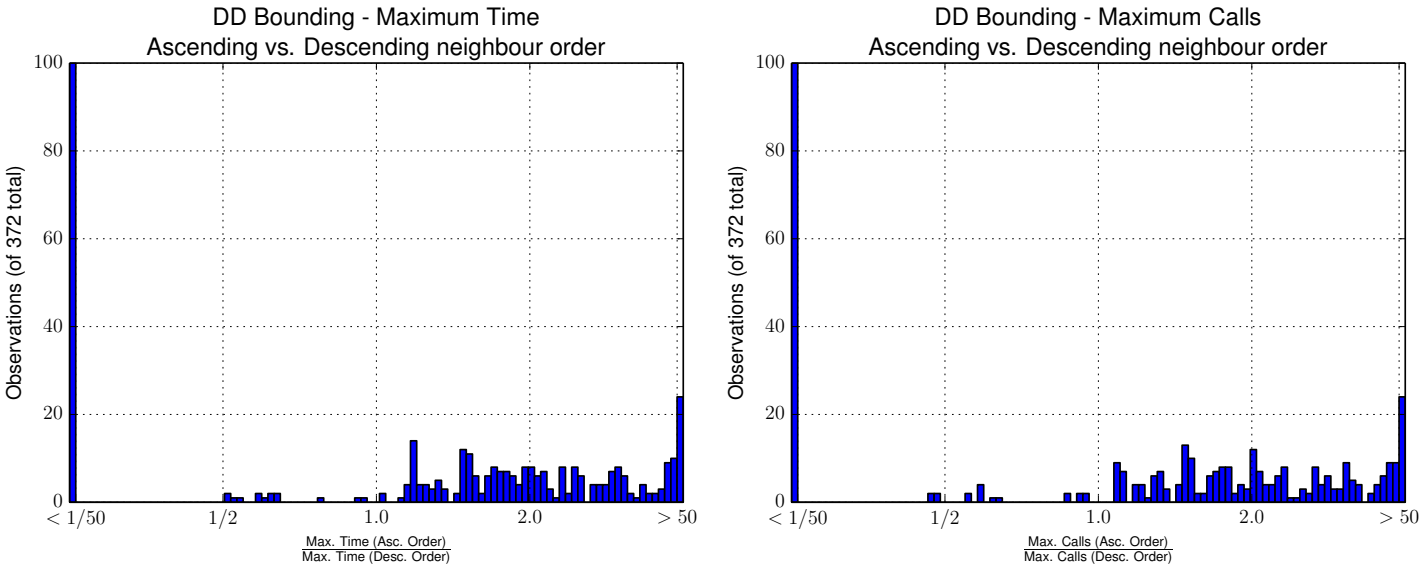
Table 4.9: DD Bounding: Summary of maximum time ratios for all aspects on all graph families

Family	Min. CD / Max. CD			Asc. Order / Desc. Order			FS Disabled / FS Enabled			RC Disabled / RC Enabled		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	16	0.067	0	0	1.471	0	0	0.979	0	0	0.915	0
Total observations:	40 pairs			32 pairs			32 pairs			32 pairs		
Covering Code Graphs	4	0.299	0	32	1.914	4	0	0.997	0	0	1.019	0
Total observations:	48 pairs			64 pairs			46 pairs			46 pairs		
Hex Rook Graphs	0	0.38	0	8	1.714	0	0	0.997	0	0	0.953	0
Total observations:	84 pairs			88 pairs			84 pairs			84 pairs		
Kneser Graphs	8	0.562	0	32	1.284	0	0	0.992	0	0	0.877	0
Total observations:	24 pairs			36 pairs			20 pairs			20 pairs		
Knight Graphs	8	0.022	0	24	20.28	0	0	1.016	0	0	0.917	0
Total observations:	52 pairs			60 pairs			48 pairs			48 pairs		
Queen Graphs	8	0.535	0	0	3.071	0	0	0.997	0	0	0.978	0
Total observations:	48 pairs			44 pairs			44 pairs			44 pairs		
Triangular Grid Graphs	30	0.005	0	4	1.958	2	0	0.993	2	0	0.905	0
Total observations:	60 pairs			48 pairs			46 pairs			45 pairs		
All Families (aggregate)	74	0.277	0	100	1.98	6	0	0.996	2	0	0.945	0
Total observations:	356 pairs			372 pairs			320 pairs			319 pairs		
All Families (equal weight)	74	0.299	0	100	1.914	6	0	0.997	2	0	0.917	0
Total observations:	356 pairs			372 pairs			320 pairs			319 pairs		



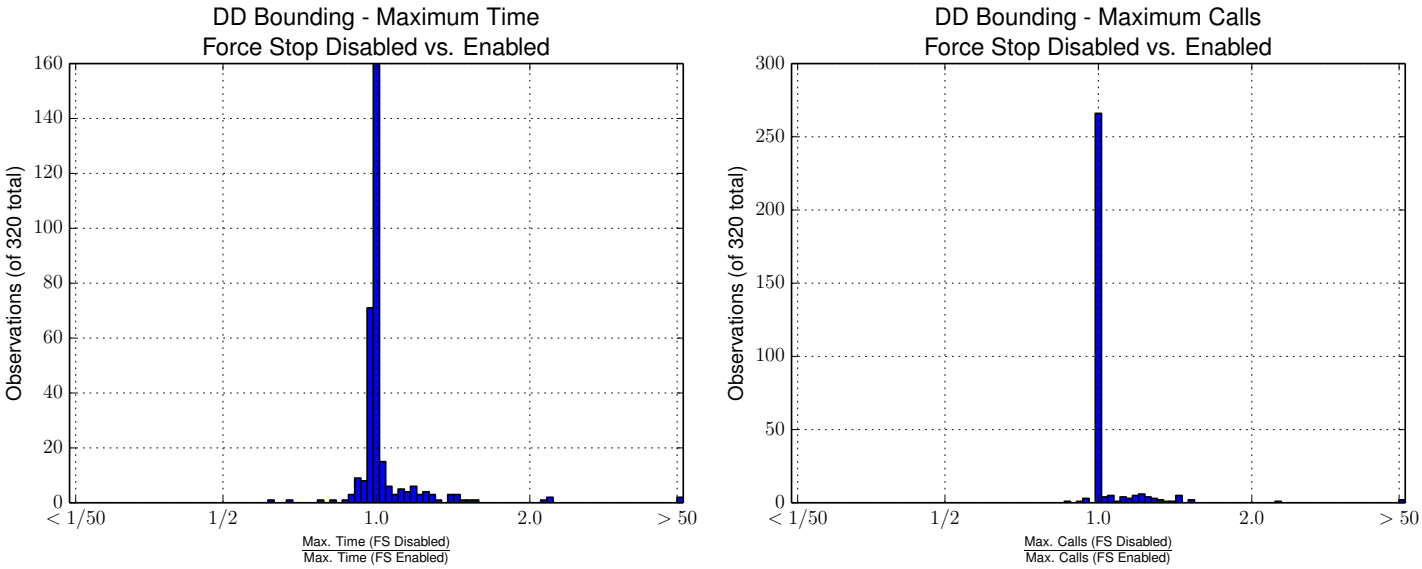
(a) Max. Time (b) Max. Calls

Figure 4.1: DD Bounding: Histogram of pairwise ratios for Min. CD vs. Max. CD vertex selection.



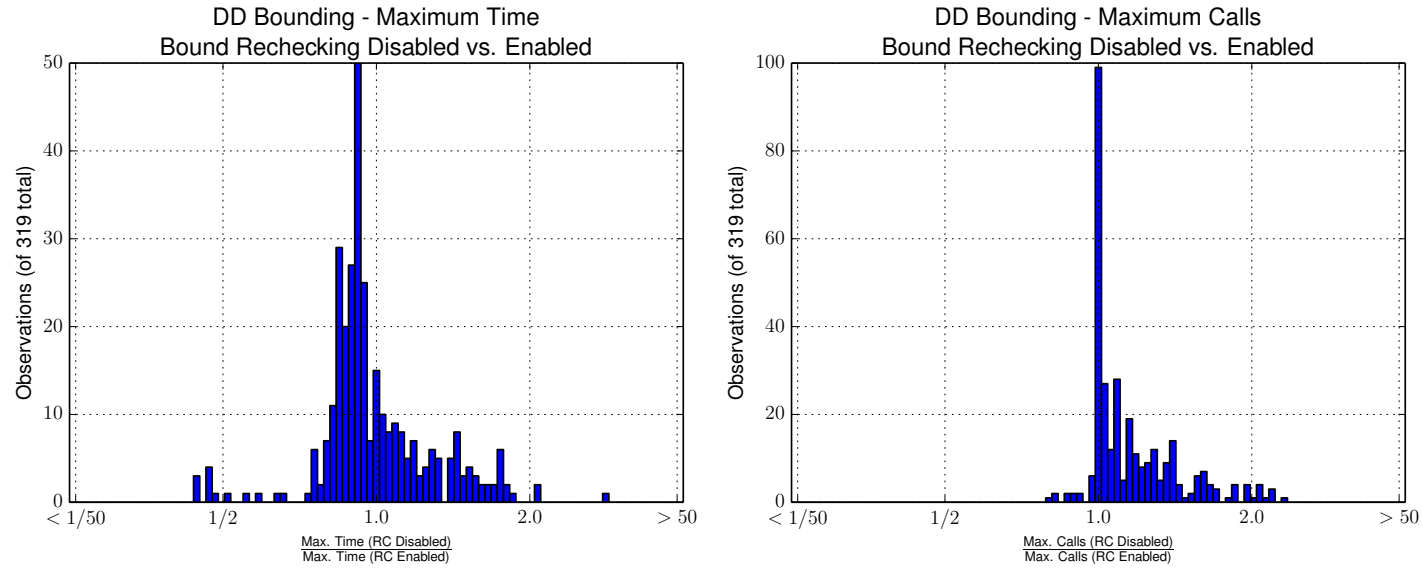
(a) Max. Time (b) Max. Calls

Figure 4.2: DD Bounding: Histogram of pairwise ratios for ascending vs. descending neighbour order.



(a) Max. Time (b) Max. Calls

Figure 4.3: DD Bounding: Histogram of pairwise ratios for force stop optimization disabled vs. enabled.



(a) Max. Time (b) Max. Calls

Figure 4.4: DD Bounding: Histogram of pairwise ratios for bound rechecking optimization disabled vs. enabled.

Table 4.10: DD Bounding: Summary of maximum total call ratios for all aspects on all graph families

Family	Min. CD / Max. CD			Asc. Order / Desc. Order			FS Disabled / FS Enabled			RC Disabled / RC Enabled		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	16	0.072	0	0	1.475	0	0	1	0	0	1.073	0
Total valid observations:	40 pairs			32 pairs			32 pairs			32 pairs		
Covering Code Graphs	4	0.278	0	32	1.816	4	0	1	0	0	1.205	0
Total valid observations:	48 pairs			64 pairs			46 pairs			46 pairs		
Hex Rook Graphs	0	0.391	0	8	1.696	0	0	1	0	0	1.037	0
Total valid observations:	84 pairs			88 pairs			84 pairs			84 pairs		
Kneser Graphs	8	0.538	0	32	1.283	0	0	1	0	0	1.01	0
Total valid observations:	24 pairs			36 pairs			20 pairs			20 pairs		
Knight Graphs	8	0.018	0	24	19.41	0	0	1	0	0	1.107	0
Total valid observations:	52 pairs			60 pairs			48 pairs			48 pairs		
Queen Graphs	8	0.518	0	0	3.196	0	0	1	0	0	1.036	0
Total valid observations:	48 pairs			44 pairs			44 pairs			44 pairs		
Triangular Grid Graphs	30	0.006	0	4	1.759	2	0	1	2	0	1.031	0
Total valid observations:	60 pairs			48 pairs			46 pairs			45 pairs		
All Families (aggregate)	74	0.267	0	100	2.014	6	0	1	2	0	1.063	0
Total valid observations:	356 pairs			372 pairs			320 pairs			319 pairs		
All Families (equal weight)	74	0.278	0	100	1.759	6	0	1	2	0	1.037	0
Total valid observations:	356 pairs			372 pairs			320 pairs			319 pairs		

4.5 Max Dominator Degree Implementations

For Algorithm 3.7, which uses a bounding condition based on max dominator degree (Theorem 3.6), 32 variants were tested. Like Algorithm 3.5, four independent aspects of the pseudocode in Section 3.4 were left undefined: the selection rule for a vertex v to be dominated at each step, the ordering of the neighbours v, the force-stop optimization and the rechecking of the bounding conditions during the loop over the neighbours of v. For the latter three aspects, the options are identical to those in Section 4.4. For the selection rule, four options were tested: choosing a vertex with minimum MDD, a vertex with maximum MDD, a vertex with minimum candidate degree and a vertex with maximum candidate degree.

4.5.1 Single Aspect Comparison

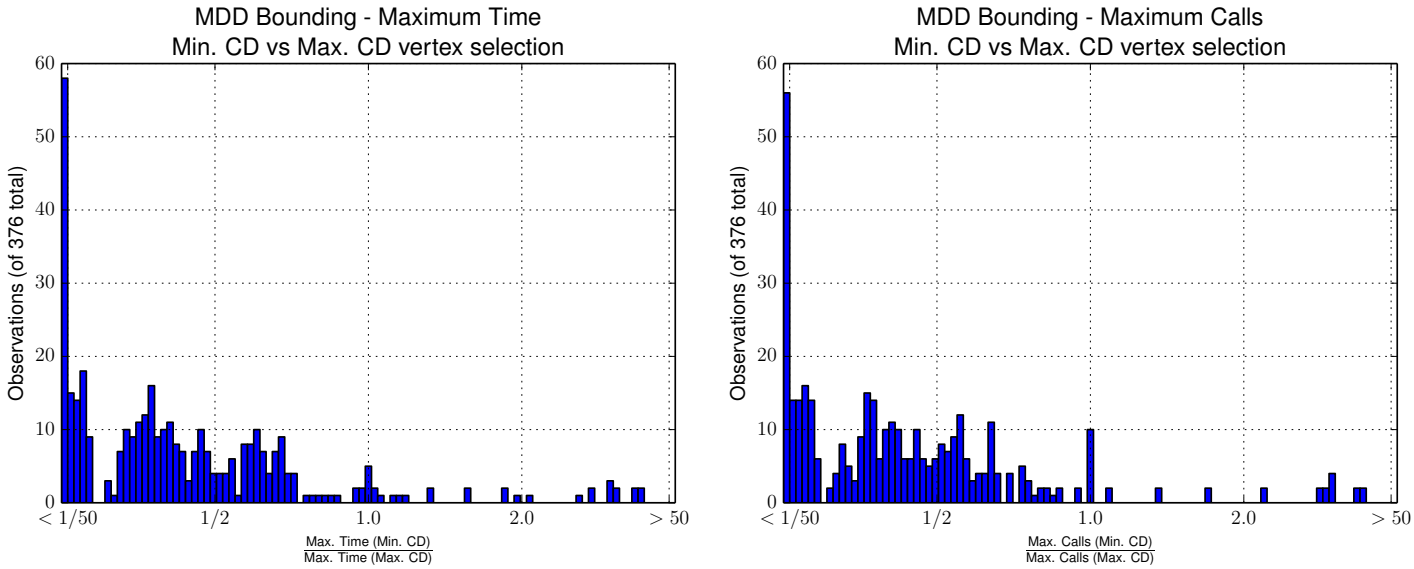
In a similar format to the histograms and tables in Section 4.4.1, Figures 4.5 - 4.13 show histograms of the multiset of pairwise ratios for each of the four aspects profiled for the MDD

bounding strategy. Tables 4.11 and 4.12 summarize the results by family. Since there were four options for the vertex selection aspect, six pairwise comparisons are presented, one for each pair out of the four options, to allow the analysis to be congruent with the analysis given in Section 4.4.1.

Overall, among the different vertex selection strategies, minimum CD generally has the advantage, with minimum MDD also having strong performance. Curiously, even though Tables 4.11 and 4.12 clearly indicate that minimum CD vertex selection has better performance on the queen graphs, the overall comparison in Section 4.6 below reveals that either minimum MDD and maximum MDD rules were superior on all of the queen graphs; this is likely due to the two MDD-based approaches 'splitting the vote', with maximum MDD being better for the smaller graphs in the family and minimum MDD being a better strategy for the larger graphs in the family, while minimum CD was, across the entire family, generally better.

The neighbour ordering comparison shows a very similar distribution to the neighbour ordering comparison on the DD bounding algorithms, with descending ordering being generally better but having a very high number of cases where it could not finish.

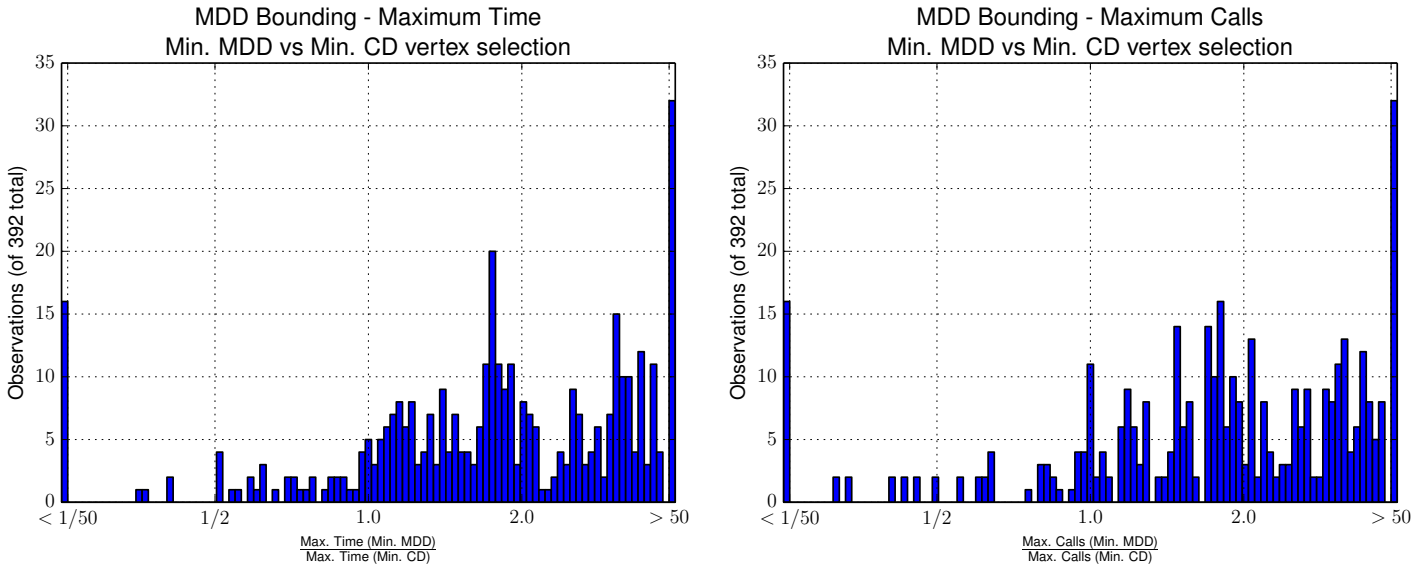
As in Section 4.4.1, the force stop optimization does not seem to have any positive effect (evidenced by the fact that the median ratio of call tree size is exactly 1), and the overhead added by the optimization outweighs the benefits in all of the aggregated statistics. Unlike in Section 4.4.1, however, enabling the bound rechecking optimization results in a significant improvement in both maximum time and maximum recursive calls, possibly because the MDD bounding conditions are more sensitive to changes between iterations of the neighbour loop.



(a) Max. Time

(b) Max. Calls

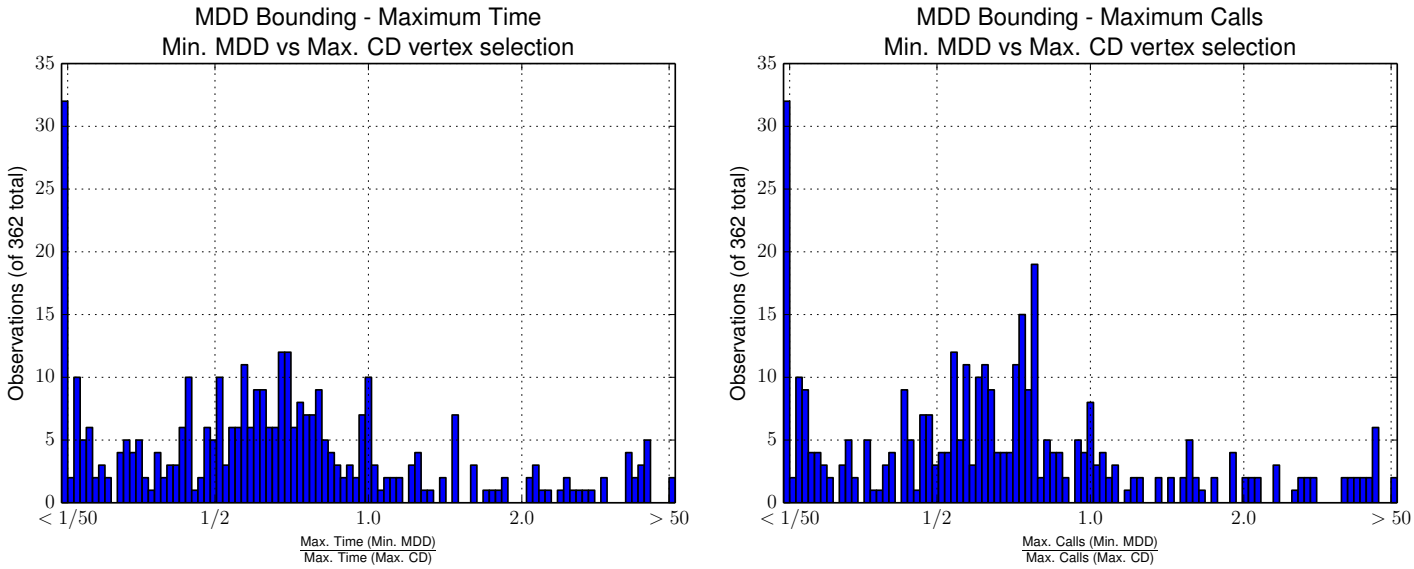
Figure 4.5: MDD Bounding: Histogram of pairwise ratios for Min. CD vs. Max. CD vertex selection.



(a) Max. Time

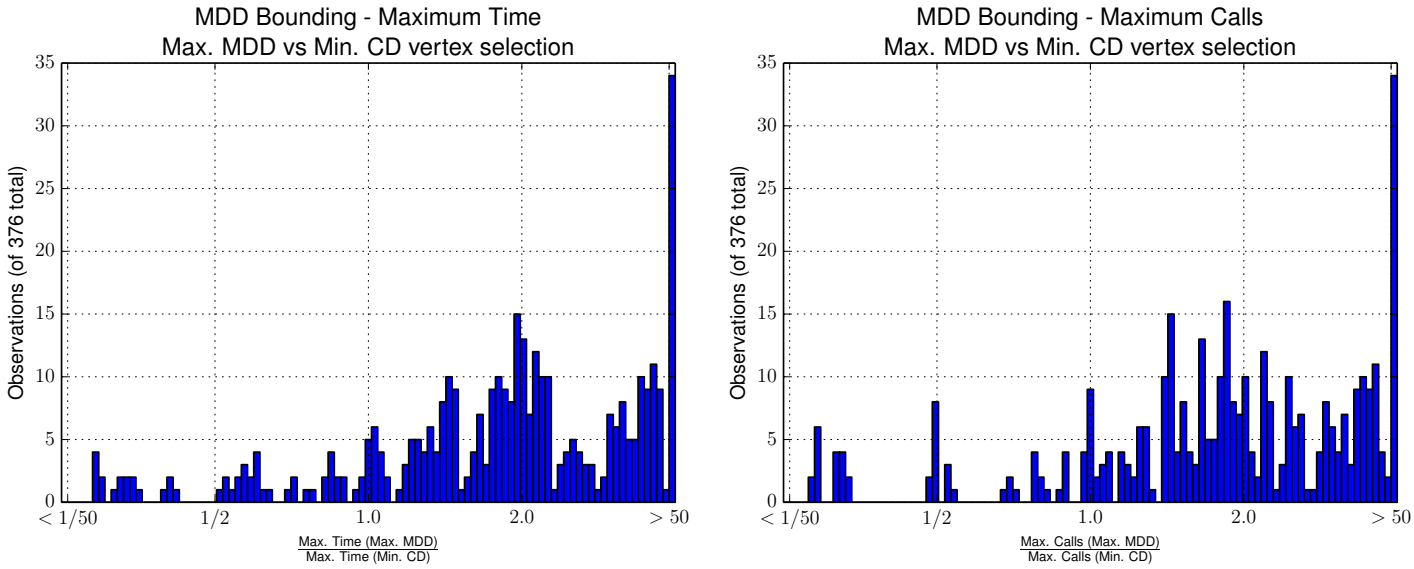
(b) Max. Calls

Figure 4.6: MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Min. CD vertex selection.



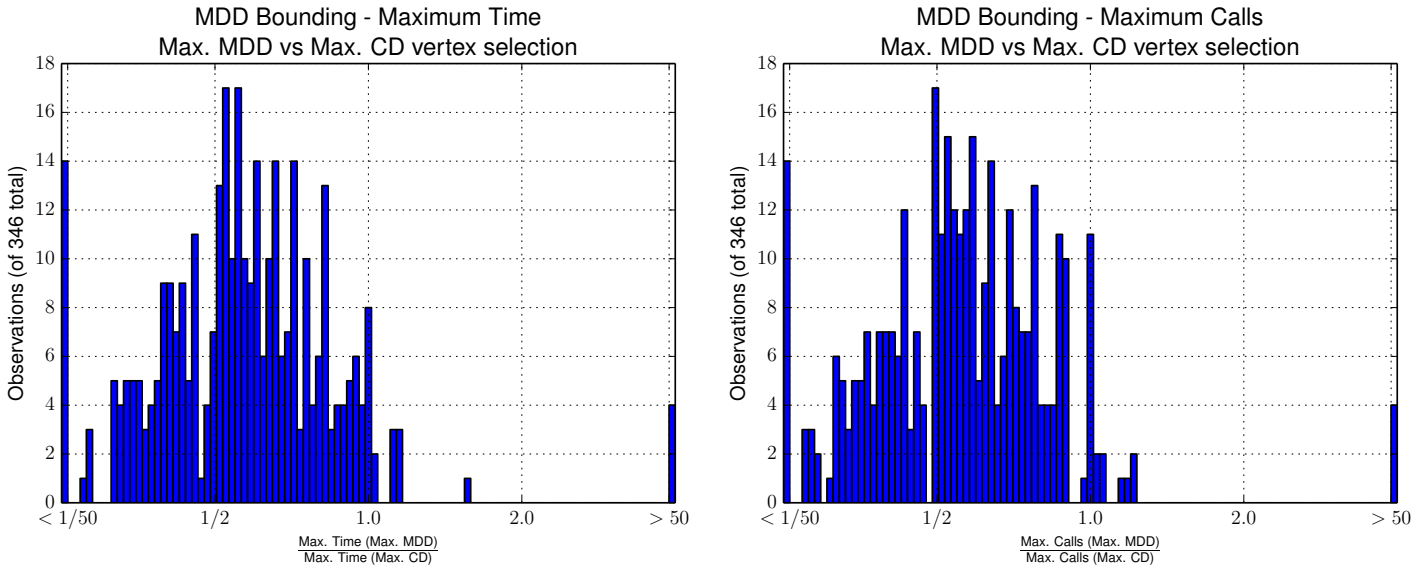
(a) Max. Time (b) Max. Calls

Figure 4.7: MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Max. CD vertex selection.



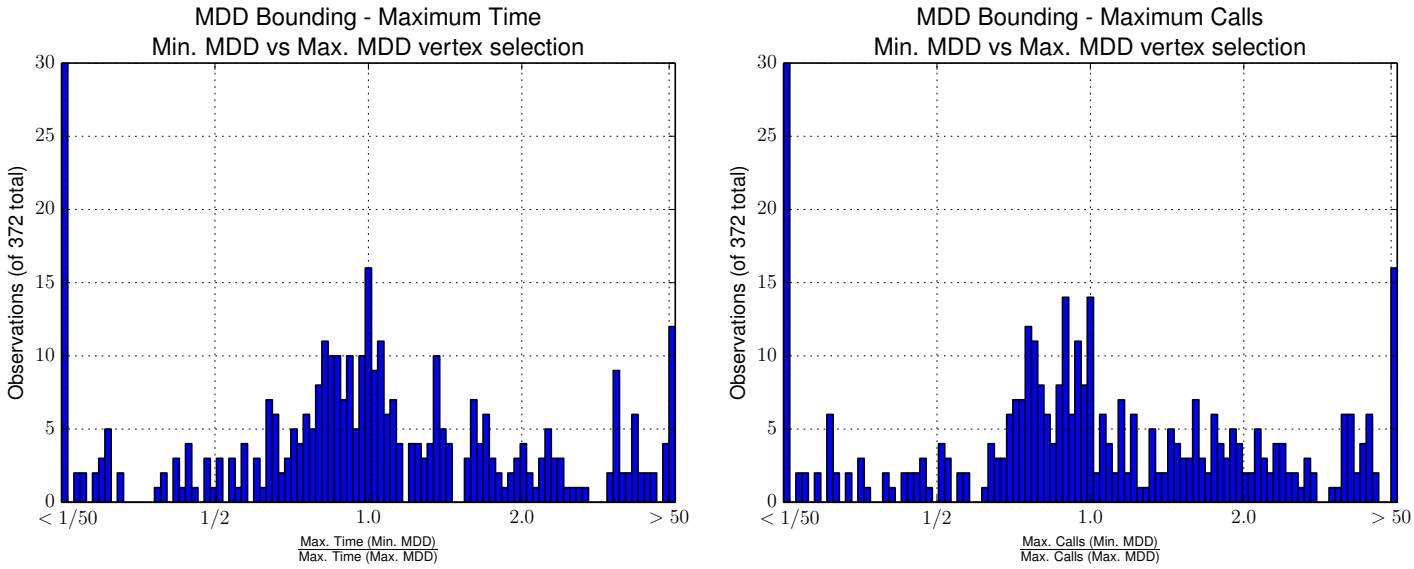
(a) Max. Time (b) Max. Calls

Figure 4.8: MDD Bounding: Histogram of pairwise ratios for Max. MDD vs. Min. CD vertex selection.



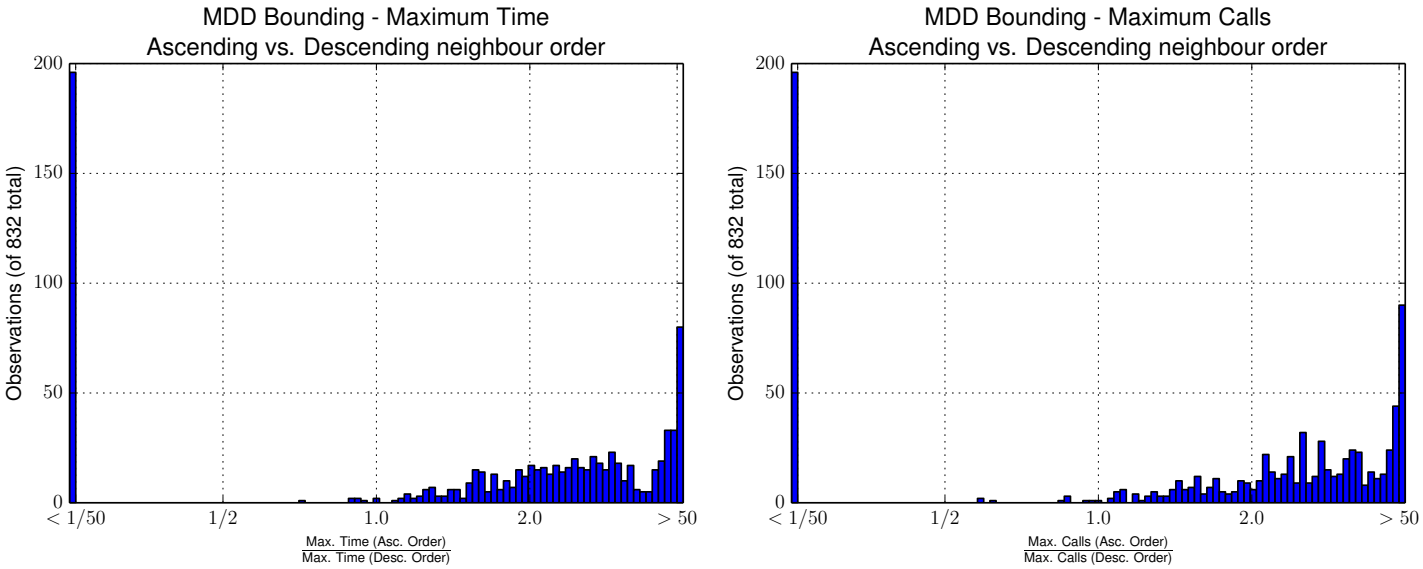
(a) Max. Time (b) Max. Calls

Figure 4.9: MDD Bounding: Histogram of pairwise ratios for Max. MDD vs. Max. CD vertex selection.



(a) Max. Time (b) Max. Calls

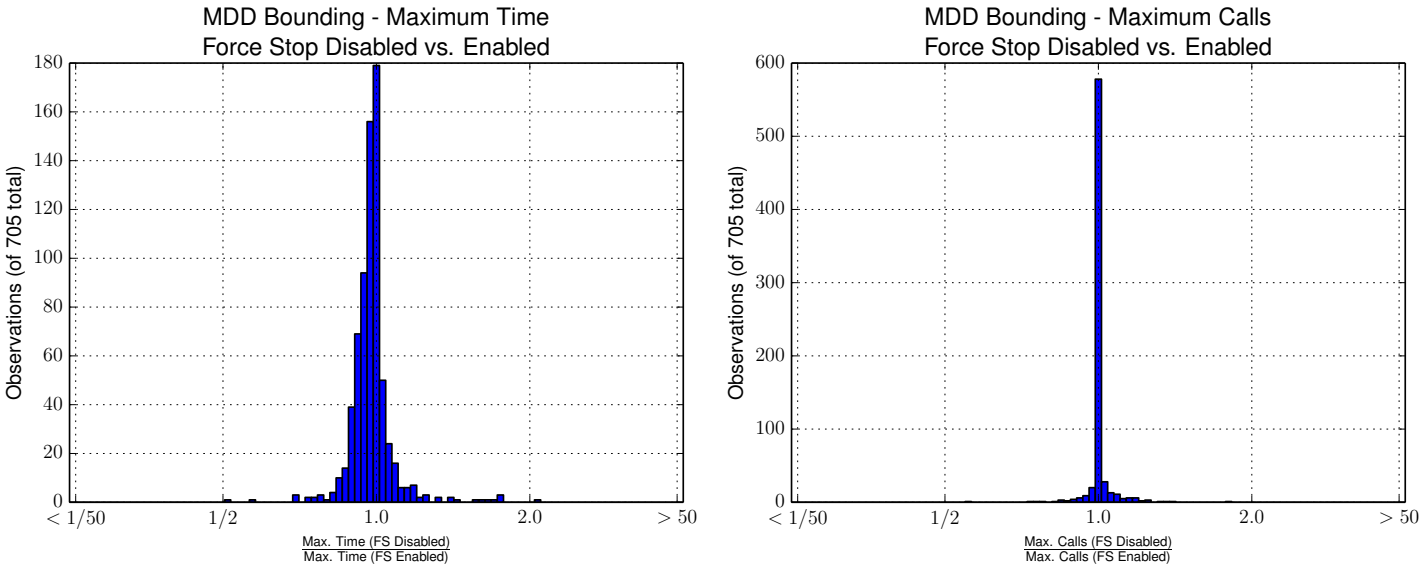
Figure 4.10: MDD Bounding: Histogram of pairwise ratios for Min. MDD vs. Max. MDD vertex selection.



(a) Max. Time

(b) Max. Calls

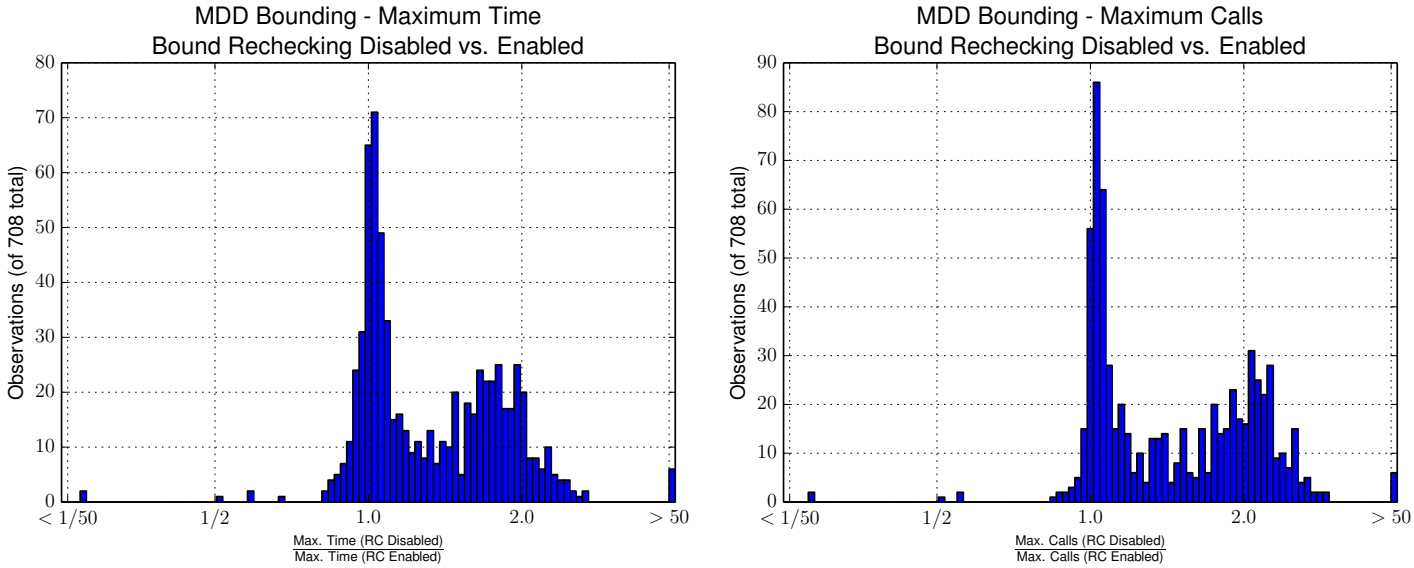
Figure 4.11: MDD Bounding: Histogram of pairwise ratios for ascending vs. descending neighbour order.



(a) Max. Time

(b) Max. Calls

Figure 4.12: MDD Bounding: Histogram of pairwise ratios for force stop optimization disabled vs. enabled.



(a) Max. Time

(b) Max. Calls

Figure 4.13: MDD Bounding: Histogram of pairwise ratios for bound rechecking optimization disabled vs. enabled.

Table 4.11: MDD Bounding: Summary of maximum time ratios for all aspects on all graph families

Family	Min. CD / Max. CD			Min. MDD / Min. CD			Min. MDD / Max. CD			Max. MDD / Min. CD			Max. MDD / Max. CD		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	16	0.244	0	0	3.886	8	8	0.64	0	0	2.324	12	4	0.509	0
Total observations:	56 pairs			56 pairs			48 pairs			56 pairs			44 pairs		
Covering Code Graphs	0	0.622	0	0	3.258	0	0	1.006	0	0	1.225	0	0	0.598	0
Total observations:	48 pairs			48 pairs			48 pairs			48 pairs			48 pairs		
Hex Rook Graphs	0	0.405	0	4	1.558	0	4	0.638	0	0	1.735	0	0	0.703	0
Total observations:	80 pairs			84 pairs			84 pairs			80 pairs			80 pairs		
Kneser Graphs	8	0.63	0	0	1.2	8	0	0.773	0	0	1.3	2	6	0.801	0
Total observations:	24 pairs			24 pairs			16 pairs			24 pairs			22 pairs		
Knight Graphs	0	0.067	0	12	5.998	0	12	0.31	0	0	5.273	0	0	0.349	0
Total observations:	52 pairs			64 pairs			64 pairs			52 pairs			52 pairs		
Queen Graphs	4	0.623	0	0	1.662	6	0	1.393	2	0	1.019	8	0	0.868	4
Total observations:	48 pairs			48 pairs			44 pairs			48 pairs			44 pairs		
Triangular Grid Graphs	16	0.205	0	0	1.103	10	6	0.417	0	0	2.42	12	4	0.415	0
Total observations:	68 pairs			68 pairs			58 pairs			68 pairs			56 pairs		
All Families (aggregate)	44	0.354	0	16	1.71	32	30	0.697	2	0	1.841	34	14	0.592	4
Total observations:	376 pairs			392 pairs			362 pairs			376 pairs			346 pairs		
All Families (equal weight)	44	0.405	0	16	1.662	32	30	0.64	2	0	1.735	34	14	0.598	4
Total observations:	376 pairs			392 pairs			362 pairs			376 pairs			346 pairs		

Family	Min. MDD / Max. MDD			Asc. Order / Desc. Order			FS Disabled / FS Enabled			RC Disabled / RC Enabled		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	4	1.132	0	0	2.709	28	0	0.952	0	0	1.236	0
Total observations:	48 pairs			108 pairs			94 pairs			94 pairs		
Covering Code Graphs	0	1.565	0	64	2.433	0	0	0.991	0	0	1.282	0
Total observations:	48 pairs			128 pairs			96 pairs			96 pairs		
Hex Rook Graphs	4	0.884	0	28	2.285	0	0	0.991	0	0	1.066	0
Total observations:	84 pairs			176 pairs			162 pairs			162 pairs		
Kneser Graphs	0	0.943	6	64	1.883	2	0	0.985	0	0	1.023	2
Total observations:	22 pairs			72 pairs			39 pairs			40 pairs		
Knight Graphs	12	1.027	0	36	13.96	0	0	0.99	0	0	1.088	0
Total observations:	64 pairs			128 pairs			110 pairs			110 pairs		
Queen Graphs	4	1.636	2	4	4.735	10	0	0.992	0	0	1.149	2
Total observations:	44 pairs			94 pairs			87 pairs			88 pairs		
Triangular Grid Graphs	6	0.856	4	0	3.099	18	0	0.961	0	0	1.372	2
Total observations:	62 pairs			126 pairs			117 pairs			118 pairs		
All Families (aggregate)	30	1	12	196	2.945	58	0	0.985	0	0	1.144	6
Total observations:	372 pairs			832 pairs			705 pairs			708 pairs		
All Families (equal weight)	30	1.027	12	196	2.709	58	0	0.99	0	0	1.149	6
Total observations:	372 pairs			832 pairs			705 pairs			708 pairs		

Table 4.12: MDD Bounding: Summary of maximum total call ratios for all aspects on all graph families

Family	Min. CD / Max. CD			Min. MDD / Min. CD			Min. MDD / Max. CD			Max. MDD / Min. CD			Max. MDD / Max. CD		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	16	0.263	0	0	3.695	8	8	0.695	0	0	1.93	12	4	0.522	0
Total observations:	56 pairs			56 pairs			48 pairs			56 pairs			44 pairs		
Covering Code Graphs	0	0.633	0	0	3.372	0	0	1	0	0	1.165	0	0	0.589	0
Total observations:	48 pairs			48 pairs			48 pairs			48 pairs			48 pairs		
Hex Rook Graphs	0	0.4	0	4	1.626	0	4	0.61	0	0	1.734	0	0	0.689	0
Total observations:	80 pairs			84 pairs			84 pairs			80 pairs			80 pairs		
Kneser Graphs	8	0.575	0	0	1.271	8	0	0.722	0	0	1.42	2	6	0.807	0
Total observations:	24 pairs			24 pairs			16 pairs			24 pairs			22 pairs		
Knight Graphs	0	0.057	0	12	6.923	0	12	0.354	0	0	5.689	0	0	0.294	0
Total observations:	52 pairs			64 pairs			64 pairs			52 pairs			52 pairs		
Queen Graphs	4	0.592	0	0	1.739	6	0	1.368	2	0	0.918	8	0	0.825	4
Total observations:	48 pairs			48 pairs			44 pairs			48 pairs			44 pairs		
Triangular Grid Graphs	16	0.224	0	0	0.988	10	6	0.462	0	0	2.444	12	4	0.396	0
Total observations:	68 pairs			68 pairs			58 pairs			68 pairs			56 pairs		
All Families (aggregate)	44	0.376	0	16	1.794	32	30	0.704	2	0	1.778	34	14	0.587	4
Total observations:	376 pairs			392 pairs			362 pairs			376 pairs			346 pairs		
All Families (equal weight)	44	0.4	0	16	1.739	32	30	0.695	2	0	1.734	34	14	0.589	4
Total observations:	376 pairs			392 pairs			362 pairs			376 pairs			346 pairs		

Family	Min. MDD / Max. MDD			Asc. Order / Desc. Order			FS Disabled / FS Enabled			RC Disabled / RC Enabled		
	0	Median	∞	0	Median	∞	0	Median	∞	0	Median	∞
Cartesian Products of Cycles	4	1.262	0	0	2.866	28	0	1	0	0	1.36	0
Total valid observations:	48 pairs			108 pairs			94 pairs			94 pairs		
Covering Code Graphs	0	1.567	0	64	2.688	0	0	1	0	0	1.314	0
Total valid observations:	48 pairs			128 pairs			96 pairs			96 pairs		
Hex Rook Graphs	4	0.871	0	28	2.614	0	0	1	0	0	1.07	0
Total valid observations:	84 pairs			176 pairs			162 pairs			162 pairs		
Kneser Graphs	0	0.873	6	64	1.964	2	0	1	0	0	1.043	2
Total valid observations:	22 pairs			72 pairs			39 pairs			40 pairs		
Knight Graphs	12	0.982	0	36	15.63	0	0	1	0	0	1.311	0
Total valid observations:	64 pairs			128 pairs			110 pairs			110 pairs		
Queen Graphs	4	1.96	2	4	5.847	10	0	1	0	0	1.418	2
Total valid observations:	44 pairs			94 pairs			87 pairs			88 pairs		
Triangular Grid Graphs	6	0.944	4	0	3.221	18	0	1	0	0	1.533	2
Total valid observations:	62 pairs			126 pairs			117 pairs			118 pairs		
All Families (aggregate)	30	0.998	12	196	3.418	58	0	1	0	0	1.281	6
Total valid observations:	372 pairs			832 pairs			705 pairs			708 pairs		
All Families (equal weight)	30	0.982	12	196	2.866	58	0	1	0	0	1.314	6
Total valid observations:	372 pairs			832 pairs			705 pairs			708 pairs		

4.6 Comparison of Framework Algorithms

Tables 4.13 - 4.19 summarize the algorithms with the best maximum running time for each of the 55 graphs in the input dataset. For each graph, the algorithm with the best maximum time from each broad variant (fixed order, DD and MDD) is shown, along with its time. All variants whose running time was within 1% of the true best time are considered tied for the

best time. The top-performing variant is shaded in gray.

For the queen graphs and the triangular grid graphs, the MDD-bounding algorithms with MDD vertex selection are unanimously the best on the tested graphs. MDD algorithms with candidate degree bounding have the best performance among the larger code graphs and most of the hex rook graphs. Although there are a few cases where the domination degree-based algorithms have decisively better performance on large graphs, the majority of cases where DD-based algorithms win outright are on very small graphs, where the overall spread of running times among the three categories is far less than one second.

As noted in Section 4.5.1, even though the MDD-bounding algorithms with the min. CD vertex selection rule have generally better performance on the family of queen graphs as a whole, Table 4.18 shows that MDD-bounding algorithms with the min. MDD and max. MDD vertex selection rules actually produce the best running times on these graphs. The high performance of the min. MDD vertex selection rule on the large queen graphs contributed to the choice of the min. MDD rule for the algorithm which was used to solve open cases of the queen problem in Chapter 5.

Table 4.13: Comparison of Framework Algorithms - Maximum Times: Covering Code Graphs

Graph	**n**	**m**	**Best Algorithm**											
			Fixed Ordering		**Domination Degree**					**Max Dominator Degree**				
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
$\text{Code}_1(2,6)$	64	192	BFS	0.267	Min. CD	desc	N	N	0.137	Min. CD	desc	N	Y	0.074
					Min. CD	desc	N	Y	0.138	Min. CD	desc	Y	Y	0.073
					Min. CD	desc	Y	N	0.137					
$\text{Code}_2(2,6)$	64	672	BFS	0.001	Min. CD	asc	Y	N	0.003	Min. CD	asc	N	Y	0.005
$\text{Code}_3(2,6)$	64	1312	NONE		Max. CD	asc	Y	Y	0.001	Max. MDD	asc	Y	Y	0.003
					Min. CD	asc	N	Y	0.001	Min. MDD	asc	Y	Y	0.003
					Min. CD	asc	Y	Y	0.001					
$\text{Code}_1(2,7)$	128	448	BFS	11.04	Min. CD	desc	N	Y	0.022	Max. MDD	desc	Y	Y	0.002
$\text{Code}_2(2,7)$	128	1792	BFS	5.16	Min. CD	desc	N	Y	5.39	Min. CD	desc	N	Y	1.763
										Min. CD	desc	Y	Y	1.768
$\text{Code}_3(2,7)$	128	4032	BFS	0.001	Max. CD	asc	N	Y	0.008	Max. MDD	asc	N	Y	0.011
					Max. CD	asc	Y	Y	0.008	Min. CD	asc	Y	Y	0.011
					Min. CD	asc	N	Y	0.008	Min. MDD	asc	Y	Y	0.011
					Min. CD	asc	Y	Y	0.008					
$\text{Code}_3(2,8)$	256	11776	BFS	0.155	Min. CD	asc	N	Y	0.427	Min. CD	asc	Y	Y	0.519
					Min. CD	asc	Y	Y	0.429					
$\text{Code}_2(3,5)$	243	6075	NONE		Min. CD	desc	N	Y	857.3	Min. CD	desc	N	Y	412.2
					Min. CD	desc	Y	Y	860.3	Min. CD	desc	Y	Y	413.8

Table 4.14: Comparison of Framework Algorithms - Maximum Times: Hex Rook Graphs

Graph	n	m	Best Algorithm												
			Fixed Ordering		Domination Degree					Max Dominator Degree					
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
HR (10)	55	495	Max. Deg.	0.01	Min. CD	asc	N	N	0.01	Min. CD	asc	Y	N	0.013
					Min. CD	asc	Y	N	0.01					
HR (11)	66	660	BFS	0.028	Min. CD	desc	N	N	0.011	Min. CD	desc	N	Y	0.01
					Min. CD	desc	Y	N	0.011	Min. CD	desc	Y	Y	0.01
HR (12)	78	858	Max. Deg.	0.306	Min. CD	asc	N	N	0.158	Min. CD	asc	N	N	0.157
										Min. CD	asc	N	Y	0.158
HR (13)	91	1092	BFS	0.463	Min. CD	desc	N	N	0.051	Min. CD	desc	N	Y	0.059
					Min. CD	desc	Y	N	0.051	Min. CD	desc	Y	Y	0.06
HR (14)	105	1365	Min. Deg.	11.68	Min. CD	asc	N	N	2.705	Min. CD	asc	N	N	2.159
					Min. CD	asc	Y	N	2.695	Min. CD	asc	N	Y	2.142
										Min. CD	asc	Y	Y	2.154
HR (15)	120	1680	BFS	15.01	Min. CD	desc	N	N	1.041	Min. CD	desc	Y	Y	0.897
					Min. CD	desc	N	Y	1.049					
					Min. CD	desc	Y	N	1.041					
HR (16)	136	2040	Max. Deg.	543.4	Min. CD	asc	N	N	55.25	Min. CD	asc	N	N	41.65
					Min. CD	asc	Y	N	55.23	Min. CD	asc	N	Y	41.87
HR (17)	153	2448	BFS	690	Min. CD	desc	N	Y	33.35	Min. CD	desc	Y	Y	20.67
					Min. CD	desc	Y	Y	33.45					
HR (18)	171	2907	NONE		Min. CD	asc	N	N	1286	Min. CD	asc	N	N	1414
					Min. CD	asc	Y	N	1283	Min. CD	asc	N	Y	1419
HR (19)	190	3420	NONE		Min. CD	desc	N	Y	878.7	Min. CD	desc	N	Y	433
					Min. CD	desc	Y	N	885	Min. CD	desc	Y	Y	433.6
HR (20)	210	3990	NONE		Min. CD	desc	N	Y	2634	Min. CD	desc	Y	Y	1145
					Min. CD	desc	Y	Y	2651					

Table 4.15: Comparison of Framework Algorithms - Maximum Times: Kneser Graphs

Graph	n	m	Best Algorithm											
			Fixed Ordering		Domination Degree					Max Dominator Degree				
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
Kneser $(8,3)$	56	280	BFS	0.018	Min. CD	asc	N	N	0.008	Min. CD	asc	N	Y	0.01
					Min. CD	asc	Y	N	0.008					
Kneser $(9,3)$	56	280	Min. Deg.	0.017	Min. CD	asc	N	N	0.007	Min. CD	asc	N	Y	0.011
					Min. CD	asc	Y	N	0.007					
Kneser $(9,4)$	126	315	NONE		Min. CD	desc	N	Y	2042	Min. CD	desc	N	Y	420.8
Kneser $(10,3)$	56	280	Max. Deg.	0.019	Min. CD	asc	N	N	0.008	Min. CD	asc	N	Y	0.008
					Min. CD	asc	Y	N	0.008	Min. CD	asc	Y	Y	0.008
Kneser $(11,3)$	56	280	Min. Deg.	0.019	Min. CD	asc	N	N	0.007	Min. CD	asc	Y	Y	0.009
					Min. CD	asc	Y	N	0.007					

Table 4.16: Comparison of Framework Algorithms - Maximum Times: Knight Graphs

Graph	n	m	Best Algorithm												
			Fixed Ordering		Domination Degree					Max Dominator Degree					
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
Knight (4)	16	24	BFS	0	Min. CD	asc	N	N	0	Max. MDD	asc	Y	N	0.001
					Min. CD	asc	Y	N	0					
Knight (5)	25	48	Min. Deg.	0	Min. CD	asc	N	N	0	Max. CD	asc	Y	Y	0.002
					Min. CD	asc	Y	N	0	Min. CD	asc	Y	Y	0.002
Knight (6)	36	80	Min. Deg.	0.001	Min. CD	asc	Y	N	0	Min. CD	asc	Y	N	0.002
Knight (7)	49	120	Min. Deg.	0.012	Min. CD	asc	N	N	0.008	Max. MDD	desc	N	Y	0.002
					Min. CD	asc	Y	N	0.008					
Knight (8)	64	168	Min. Deg.	0.286	Max. CD	desc	Y	Y	0.066	Max. MDD	desc	Y	Y	0.008
Knight (9)	81	224	Min. Deg.	0.559	Min. CD	desc	N	Y	0.03	Min. CD	desc	N	Y	0.014
Knight (10)	100	288	Min. Deg.	8.263	Min. CD	desc	N	Y	0.067	Min. CD	desc	N	Y	0.03
Knight (11)	121	360	Min. Deg.	3017	Min. CD	desc	N	N	19.81	Min. CD	desc	N	Y	6
					Min. CD	desc	Y	N	19.81					

Table 4.17: Comparison of Framework Algorithms - Maximum Times: Cartesian Products of Cycles

Graph	n	m	Best Algorithm											
			Fixed Ordering		Domination Degree					Max Dominator Degree				
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
$C_8 \square C_8$	64	128	BFS	0.725	Min. CD	asc	N	N	0.712	Min. CD	asc	N	N	0.352
$C_9 \square C_9$	81	162	BFS	0.679	Min. CD	desc	N	N	0.182	Min. CD	desc	N	Y	0.026
					Min. CD	desc	Y	N	0.183					
$C_{10} \square C_{10}$	100	200	BFS	0.71	Min. CD	desc	N	N	1.722	Min. CD	desc	N	Y	0.084
										Min. CD	desc	Y	Y	0.084
$C_{11} \square C_{11}$	121	242	BFS	449.6	Min. CD	desc	N	N	154	Min. CD	desc	N	Y	4.821
$C_{12} \square C_{12}$	144	288	NONE		Min. CD	asc	N	N	3926	Min. CD	desc	N	Y	180.6
$C_{13} \square C_{13}$	169	338	NONE		NONE					Min. CD	desc	N	Y	5480
$C_{14} \square C_{14}$	196	392	NONE		NONE					NONE				
$C_{15} \square C_{15}$	225	450	NONE		NONE					NONE				

Table 4.18: Comparison of Framework Algorithms - Maximum Times: Queen Graphs

Graph	n	m	Best Algorithm											
			Fixed Ordering		Domination Degree					Max Dominator Degree				
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
Queen (10)	100	1470	Max. Deg.	0.172	Max. CD	desc	N	Y	0.016	Max. MDD	desc	N	Y	0.014
			Min. Deg.	0.171	Max. CD	desc	Y	Y	0.016	Max. MDD	desc	Y	Y	0.014
Queen (11)	121	1980	Max. Deg.	0.281	Max. CD	desc	N	Y	0.018	Max. MDD	desc	N	Y	0.01
Queen (12)	144	2596	Max. Deg.	12.13	Max. CD	desc	N	Y	0.438	Max. MDD	desc	N	Y	0.166
					Max. CD	desc	Y	Y	0.439					
Queen (13)	169	3328	BFS	216.7	Min. CD	asc	N	N	10.28	Min. MDD	desc	N	Y	3.412
					Min. CD	asc	Y	N	10.27					
Queen (14)	196	4186	BFS	6646	Min. CD	asc	N	N	177.1	Min. MDD	desc	N	Y	96.59
			Min. Deg.	6693	Min. CD	asc	Y	N	177.1	Min. MDD	desc	Y	Y	96.74
Queen (15)	225	5180	NONE		Min. CD	asc	N	N	4277	Min. MDD	desc	N	Y	3069
					Min. CD	asc	Y	N	4302	Min. MDD	desc	Y	Y	3075

Table 4.19: Comparison of Framework Algorithms - Maximum Times: Triangular Grid Graphs

Graph	n	m	Best Algorithm												
			Fixed Ordering		Domination Degree					Max Dominator Degree					
			Vertex Order	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)	Selection Rule	N. Order	FS	RC	Time (s)
TG (11)	66	165	Min. Deg.	0.132	Min. CD	asc	N	N	0.138	Max. MDD	desc	N	Y	0.02
					Min. CD	asc	Y	N	0.139					
TG (12)	78	198	Min. Deg.	0.655	Min. CD	desc	N	N	0.301	Min. MDD	desc	N	Y	0.058
					Min. CD	desc	Y	N	0.302					
TG (13)	91	234	Min. Deg.	6.038	Min. CD	desc	N	N	1.253	Min. MDD	desc	N	Y	0.147
TG (14)	105	273	Min. Deg.	82.84	Min. CD	desc	N	N	9.839	Min. MDD	desc	N	Y	0.278
TG (15)	120	315	Min. Deg.	249.8	Min. CD	asc	N	N	40	Min. MDD	desc	N	Y	2.479
					Min. CD	asc	Y	N	40.22					
TG (16)	136	360	Min. Deg.	6797	Min. CD	asc	N	N	568	Min. MDD	desc	N	Y	22.3
					Min. CD	asc	Y	N	570.8					
TG (17)	153	408	NONE		Min. CD	desc	N	N	3121	Min. MDD	desc	N	Y	112.9
TG (18)	171	459	NONE		NONE					Min. MDD	desc	N	Y	842.9
										Min. MDD	desc	Y	Y	845.5
TG (19)	190	513	NONE		NONE					NONE				

4.7 Comparison with SageMath

The SageMath suite [1] provides a dominating set solver which uses mixed integer-linear programming to find a minimum dominating set of a graph. The integer program used to solve the dominating set problem is given in Section 1.3. The solver does not have options to generate all dominating sets of a given size. SageMath supports several integer-linear programming solvers, but uses the GLPK solver by default. For these experiments, SageMath with GLPK was used to provide a reference point for comparison with the family of new algorithms created during this research.

The SageMath dominating set solver was run on all of the inputs used in the previous experiments and the minimum, maximum and average times of SageMath on the 10 permutations of each input graph was computed. Since the SageMath solver does not provide information on call tree size, and since the structure of the underlying algorithm is not comparable to Framework 3.1, the only meaningful comparison between the two is running time. The times were calculated from the beginning of the dominating set computation (after the input graph had already been read) until the computed minimum dominating set was re-

turned. Tables 4.20 - 4.26 show the maximum time of SageMath on each input graph along with the best algorithm based on Framework 3.1 and its maximum time. The overall best algorithm is highlighted in gray.

Table 4.20: SageMath vs. Framework 3.1 - Maximum Times: Covering Code Graphs

Graph	**n**	**m**	γ	**SageMath**	**Framework 3.1**	
				Max. Time (s)	**Variant**	**Max. Time (s)**
$\text{Code}_1(2,6)$	64	192	12	0.27	MDD Bounding (Min. CD, desc, FS, RC)	0.073
$\text{Code}_2(2,6)$	64	672	4	0.041	Fixed Order (BFS)	0.001
$\text{Code}_3(2,6)$	64	1312	2	0.055	DD Bounding (Max. CD, asc, FS, RC)	0.001
$\text{Code}_1(2,7)$	128	448	16	0.021	MDD Bounding (Max. MDD, desc, FS, RC)	0.002
$\text{Code}_2(2,7)$	128	1792	7	21.25	MDD Bounding (Min. CD, desc, no FS, RC)	1.763
$\text{Code}_3(2,7)$	128	4032	2	0.035	Fixed Order (BFS)	0.001
$\text{Code}_3(2,8)$	256	11776	4	13.24	Fixed Order (BFS)	0.155
$\text{Code}_2(3,5)$	243	6075	8	N/A	MDD Bounding (Min. CD, desc, no FS, RC)	412.2

Table 4.21: SageMath vs. Framework 3.1 - Maximum Times: Hex Rook Graphs

Graph	**n**	**m**	γ	**SageMath**	**Framework 3.1**	
				Max. Time (s)	**Variant**	**Max. Time (s)**
HR(10)	55	495	5	0.223	DD Bounding (Min. CD, asc, no FS, no RC)	0.01
HR(11)	66	660	5	0.8	MDD Bounding (Min. CD, desc, no FS, RC)	0.01
HR(12)	78	858	6	4.033	MDD Bounding (Min. CD, asc, no FS, no RC)	0.157
HR(13)	91	1092	6	3.799	DD Bounding (Min. CD, desc, FS, no RC)	0.051
HR(14)	105	1365	7	127.1	MDD Bounding (Min. CD, asc, no FS, RC)	2.142
HR(15)	120	1680	7	78.45	MDD Bounding (Min. CD, desc, FS, RC)	0.897
HR(16)	136	2040	8	N/A	MDD Bounding (Min. CD, asc, no FS, no RC)	41.65
HR(17)	153	2448	8	6438	MDD Bounding (Min. CD, desc, FS, RC)	20.67
HR(18)	171	2907	9	N/A	DD Bounding (Min. CD, asc, FS, no RC)	1283
HR(19)	190	3420	9	N/A	MDD Bounding (Min. CD, desc, no FS, RC)	433
HR(20)	210	3990	9	N/A	MDD Bounding (Min. CD, desc, FS, RC)	1145

Table 4.22: SageMath vs. Framework 3.1 - Maximum Times: Kneser Graphs

Graph	**n**	**m**	γ	**SageMath**	**Framework 3.1**	
				Max. Time (s)	**Variant**	**Max. Time (s)**
Kneser(8,3)	56	280	7	0.142	DD Bounding (Min. CD, asc, FS, no RC)	0.008
Kneser(9,3)	56	280	7	0.106	DD Bounding (Min. CD, asc, no FS, no RC)	0.007
Kneser(9,4)	126	315	26	5677	MDD Bounding (Min. CD, desc, no FS, RC)	420.8
Kneser(10,3)	56	280	7	0.111	DD Bounding (Min. CD, asc, no FS, no RC)	0.008
Kneser(11,3)	56	280	7	0.116	DD Bounding (Min. CD, asc, no FS, no RC)	0.007

Table 4.23: SageMath vs. Framework 3.1 - Maximum Times: Knight Graphs

Graph	n	m	γ	SageMath	Framework 3.1	
				Max. Time (s)	Variant	Max. Time (s)
Knight (4)	16	24	4	0.01	DD Bounding (Min. CD, asc, FS, no RC)	0
Knight (5)	25	48	5	0.012	DD Bounding (Min. CD, asc, FS, no RC)	0
Knight (5)	25	48	5	0.012	DD Bounding (Min. CD, asc, no FS, no RC)	0
Knight (6)	36	80	8	0.012	DD Bounding (Min. CD, asc, FS, no RC)	0
Knight (7)	49	120	10	0.014	MDD Bounding (Max. MDD, desc, no FS, RC)	0.002
Knight (8)	64	168	12	0.014	MDD Bounding (Max. MDD, desc, FS, RC)	0.008
Knight (9)	81	224	14	0.03	MDD Bounding (Min. CD, desc, no FS, RC)	0.014
Knight (10)	100	288	16	0.18	MDD Bounding (Min. CD, desc, no FS, RC)	0.03
Knight (11)	121	360	21	0.713	MDD Bounding (Min. CD, desc, no FS, RC)	6

Table 4.24: SageMath vs. Framework 3.1 - Maximum Times: Cartesian Products of Cycles

Graph	n	m	γ	SageMath	Framework 3.1	
				Max. Time (s)	Variant	Max. Time (s)
$C_8 \square C_8$	64	128	16	0.817	MDD Bounding (Min. CD, asc, no FS, no RC)	0.352
$C_9 \square C_9$	81	162	18	0.121	MDD Bounding (Min. CD, desc, no FS, RC)	0.026
$C_{10} \square C_{10}$	100	200	20	0.017	MDD Bounding (Min. CD, desc, FS, RC)	0.084
$C_{11} \square C_{11}$	121	242	27	2.501	MDD Bounding (Min. CD, desc, no FS, RC)	4.821
$C_{12} \square C_{12}$	144	288	32	20.46	MDD Bounding (Min. CD, desc, no FS, RC)	180.6
$C_{13} \square C_{13}$	169	338	38	378.4	MDD Bounding (Min. CD, desc, no FS, RC)	5480
$C_{14} \square C_{14}$	196	392	42	36.85	NONE	*N/A*
$C_{15} \square C_{15}$	225	450	45	0.031	NONE	*N/A*

Table 4.25: SageMath vs. Framework 3.1 - Maximum Times: Queen Graphs

Graph	n	m	γ	SageMath	Framework 3.1	
				Max. Time (s)	Variant	Max. Time (s)
Queen (10)	100	1470	5	2.74	MDD Bounding (Max. MDD, desc, no FS, RC)	0.014
Queen (11)	121	1980	5	7.488	MDD Bounding (Max. MDD, desc, no FS, RC)	0.01
Queen (12)	144	2596	6	113.8	MDD Bounding (Max. MDD, desc, no FS, RC)	0.166
Queen (13)	169	3328	7	223.4	MDD Bounding (Min. MDD, desc, no FS, RC)	3.412
Queen (14)	196	4186	8	N/A	MDD Bounding (Min. MDD, desc, no FS, RC)	96.59
Queen (15)	225	5180	9	N/A	MDD Bounding (Min. MDD, desc, no FS, RC)	3069

Table 4.26: SageMath vs. Framework 3.1 - Maximum Times: Triangular Grid Graphs

Graph	n	m	γ	SageMath	Framework 3.1	
				Max. Time (s)	Variant	Max. Time (s)
TG (11)	66	165	13	0.019	MDD Bounding (Max. MDD, desc, no FS, RC)	0.02
TG (12)	78	198	15	0.019	MDD Bounding (Min. MDD, desc, no FS, RC)	0.058
TG (13)	91	234	17	0.029	MDD Bounding (Min. MDD, desc, no FS, RC)	0.147
TG (14)	105	273	19	0.036	MDD Bounding (Min. MDD, desc, no FS, RC)	0.278
TG (15)	120	315	21	0.051	MDD Bounding (Min. MDD, desc, no FS, RC)	2.479
TG (16)	136	360	24	0.095	MDD Bounding (Min. MDD, desc, no FS, RC)	22.3
TG (17)	153	408	27	0.159	MDD Bounding (Min. MDD, desc, no FS, RC)	112.9
TG (18)	171	459	30	0.554	MDD Bounding (Min. MDD, desc, no FS, RC)	842.9
TG (19)	190	513	33	1.129	NONE	*N/A*

4.8 Choosing Representative Algorithms

The set of 52 algorithms covered by the experiments in this chapter is unwieldy in a practical setting, since it is not feasible to investigate the performance of all variants before undertaking a dominating set computation. The overall goal of the experiments was to produce a small set of solvers which, together, would be useful for future research. This section describes the selection of several high quality representative algorithms from the set of 51 variants of Framework 3.1 covered by the experiments. Section 4.8.1 ranks the variants by their overall performance across all families, and Section 4.8.2 contains a discussion of the performance of the different algorithms on each of the graph families studied. Based on the data in both sections, the following variants were chosen as representatives and were incorporated into the `unidom` program described in Chapter 6.

Fixed Order (BFS)
DD Bounding (Min. CD, asc, FS, no RC)
DD Bounding (Min. CD, desc, FS, no RC)
MDD Bounding (Min. CD, desc, no FS, RC)
MDD Bounding (Min. MDD, desc, no FS, RC)

Let $\mathcal{A}$ be the set of all 52 algorithms tested by the experiments in this chapter (SageMath plus 51 variants of Framework 3.1). Let $\mathcal{G}$ be the set of all initial input graphs for the experiment, such that for each graph $G \in \mathcal{G}$, ten permutations of G were tested against all algorithms in $\mathcal{A}$. For $A \in \mathcal{A}, G \in \mathcal{G}$, let MaxTime$(G, A)$ denote the maximum running time of algorithm A over all of the 10 permutations of graph G used in the experiment. If A did not finish on any of the permutations of G, then define MaxTime$(G, A) = \infty$.

Several caveats are needed before a meaningful comparison of algorithms can be made. First, while the input dataset for the experiments contained 55 graphs, many of these graphs were solved in a very small amount of time by one of the tested solvers. Graphs for which a solver was able to solve each of the 10 tested permutations in a small amount of time may be considered 'easy' for the purposes of the domination problem, and while the ability of a solver

to handle easy graphs is important, the design of an algorithm for a computational search to solve an open problem (such as computing the domination number of Queen (20)) would likely benefit more from the experimental data for more difficult graphs. In this section, a graph G in the input dataset is defined to be *moderately difficult* if

$$\min_{A \in \mathcal{A}} [\text{MaxTime}(G, A)] \quad \geq \quad 0.5 \text{ seconds.}$$

Some of the comparisons in this section, particularly in Section 4.8.2, are constrained to the set of 20 moderately difficult graphs instead of the complete input dataset of 55 graphs. The set of experimental data for the 20 moderately difficult graphs is still quite large, since it comprises running times for 52 algorithms on 200 permutations.

Second, a meaningful metric is needed to compare the running times of different algorithms across multiple graphs. For example, to characterize the performance of an algorithm A over the entire input dataset $\mathcal{G}$, the average of $\text{MaxTime}(G, A)$ across all graphs $G \in \mathcal{G}$ would not be a reliable metric, since if A failed to finish on one graph, the average would equal infinity. Additionally, even in cases where A finished on all graphs, the average would be unreasonably biased by large running times. We propose a statistic called the *average time fraction* to track the performance of a particular algorithm over a set of graphs, which remedies both of the issues which affect the simple average of running times. Although this metric is not a panacea, it can be useful to assess the comparative performance of algorithms, especially within a group of similar graphs.

For a particular graph G and algorithm A, define $\text{MaxTimeFraction}(G, A)$ to be the ratio of the best maximum time of any algorithm in $\mathcal{A}$ on G to the maximum time of A on G:

$$\text{MaxTimeFraction}(G, A) = \frac{\min_{A' \in \mathcal{A}} \text{MaxTime}(G, A')}{\text{MaxTime}(G, A)}.$$

The peculiar ordering of numerator and denominator ensures that the resulting quantity will

be in the range $[0, 1]$ and that, if algorithm A did not finish in graph G, MaxTimeFraction$(G, A) = 0$ (since MaxTime$(G, A) = \infty$ in such cases). Larger values of MaxTimeFraction(G, A) indicate that the maximum time of algorithm A on G was close to the best time over all algorithms.

Define the *average time fraction* of A with respect to a collection of graphs $\mathcal{G}$ and algorithms $\mathcal{A}$ to be

$$\text{AvgFrac}(A, \mathcal{G}, \mathcal{A}) = \frac{1}{|\mathcal{G}|} \sum_{G \in \mathcal{G}} \text{MaxTimeFraction}(G, A).$$

Since MaxTimeFraction(G, A) has a normalized range, AvgFrac$(A, \mathcal{G}, \mathcal{A})$ can be used as a general measurement of the performance of an algorithm $A \in \mathcal{A}$ across the entire dataset.

4.8.1 Overall Variant Comparison

The goal of the experiments documented in this chapter was to examine the impact of different implementation decisions for Framework 3.1 with an eye toward producing high quality general purpose dominating set algorithms. Chapter 6 describes the finished general purpose solver, which includes several different implementations of algorithms based on Framework 3.1.

The tables in Sections 4.6 and 4.7 show that none of the tested algorithms stand out as the single best option, but it is possible to narrow down the set of 51 tested variants to a handful of high quality algorithms.

Table 4.6 shows the 'best' algorithm for each input graph, and one option for choosing a high quality representative solver would be to simply take the set of algorithms which appear in Table 4.6 most frequently. However, this does not account for cases where a particular algorithm has good performance in a small number of cases but otherwise is very uncompetitive. For example, it is possible that for all cases in Table 4.6, the second-place

algorithm is the same, and would therefore have very competitive performance on the entire input dataset. Measuring the overall value of each algorithm across the entire input dataset is difficult: simple metrics like the average running time are not useful due to the high variation of running times across the dataset. The average time fraction proposed in the previous section may provide a solution to this issue.

Table 4.27 shows the best 10 average time fractions of algorithms in $\mathcal{A}$ on all graphs, and Table 4.28 shows the best 10 average time fractions of algorithms in $\mathcal{A}$ on the 20 'moderately difficult' graphs for which every solver had a maximum running time of at least 0.5 seconds.

Table 4.27: Best 10 average maximum time fractions of tested algorithms on the entire input dataset.

Algorithm	**Avg. Fraction**
SageMath	0.344
DD Bounding (Min. CD, asc, FS, no RC)	0.337
DD Bounding (Min. CD, asc, no FS, no RC)	0.329
MDD Bounding (Min. CD, asc, no FS, RC)	0.300
MDD Bounding (Min. CD, asc, FS, RC)	0.297
MDD Bounding (Min. CD, asc, no FS, no RC)	0.296
DD Bounding (Min. CD, asc, no FS, RC)	0.293
MDD Bounding (Min. CD, asc, FS, no RC)	0.292
DD Bounding (Min. CD, asc, FS, RC)	0.289
MDD Bounding (Min. CD, desc, no FS, RC)	0.275

Table 4.28: Best 10 average maximum time fractions of tested algorithms on the 20 moderately difficult graphs in the input dataset.

Algorithm	**Avg. Fraction**
MDD Bounding (Min. CD, desc, no FS, RC)	0.389
MDD Bounding (Min. CD, desc, FS, RC)	0.385
MDD Bounding (Min. CD, asc, FS, RC)	0.361
SageMath	0.360
MDD Bounding (Min. CD, asc, no FS, RC)	0.360
MDD Bounding (Min. CD, asc, no FS, no RC)	0.360
MDD Bounding (Min. CD, asc, FS, no RC)	0.351
MDD Bounding (Min. MDD, desc, no FS, RC)	0.322
MDD Bounding (Min. MDD, desc, FS, RC)	0.317
DD Bounding (Min. CD, asc, FS, no RC)	0.300

4.8.2 Comparison of Variants by Graph Family

All of the graph families used in the input dataset have open questions regarding domination number, and the data collected in these experiments can be used to guide future computational research into the domination numbers of open cases. This section contains a summary of the data for moderately difficult cases in each of the graph families, along with recommendations of which algorithms to use in future computational research.

Many of the graphs in the input dataset were solved extremely quickly by one or more variants. Although these cases are useful for designing a general purpose solver, an algorithm's performance on easier cases does not necessarily correlate to its performance on the very long searches needed to solve open problems. For the data in this section, only those graphs for which every variant of Framework 3.1 had a maximum running time across all permutations of at least 0.5 seconds were considered. These graphs can be considered to be the 'moderately difficult' members of the family, and are more likely to indicate the suitability of an algorithm for hard cases than the graphs with extremely fast solution times since the impact of constant factors (such as preprocessing overhead and 'noise' in the timings) is minimized. Within each family of graphs, a small set of candidate algorithms was chosen by taking all algorithms which displayed good performance on at least one of the graphs in the family. The exact parameters for the selection varied between families of graphs due to differing spreads of running times.

Of the covering code graphs, only two graphs were moderately difficult, $\text{Code}_2(2, 7)$ and $\text{Code}_2(3, 5)$. The set of candidate algorithms was chosen to contain all variants whose running time was at most 150% of the best running time for at least one of the two graphs. This criteria produces a set of five candidates, of which the MDD Bounding (Min. CD, desc, no FS, RC) and MDD Bounding (Min. CD, desc, FS, RC) variants have the best performance on both graphs, with a slight advantage for the MDD Bounding (Min. CD, desc, no FS, RC) variant. For this family, the MDD Bounding (Min. CD, desc, no FS, RC)

is therefore chosen as the best representative algorithm. Table 4.29 shows the maximum times for all five candidates, along with the average time fraction for each candidate across both graphs.

Table 4.29: Maximum times of Framework 3.1 variants on Covering Code graphs.

Algorithm	**Time (seconds)**		**Avg.**
	$\mathrm{Code}_2(2,7)$	$\mathrm{Code}_2(3,5)$	**Frac.**
MDD Bounding (Max. MDD, desc, FS, RC)	2.452	813.5	0.613
MDD Bounding (Max. MDD, desc, no FS, RC)	2.427	818.4	0.615
MDD Bounding (Min. CD, desc, FS, RC)	1.768	413.8	0.996
MDD Bounding (Min. CD, desc, FS, no RC)	3.616	551.7	0.617
MDD Bounding (Min. CD, desc, no FS, RC)	1.763	412.2	1.000
SageMath	21.25	–	0.041

The Hex Rook graphs HR(14) - HR(20) were all classified as moderately difficult. On these graphs, a large number of variants were clustered very close to the best running time. To choose a set of candidates, all variants whose maximum running times were at most 101% of the best maximum running time on at least one of these graphs were chosen, producing a set of 7 candidates based on Framework 3.1. Table 4.30 shows the resulting comparison, with the results for SageMath also shown, even though SageMath would not qualify as a candidate by the rule above, along with the average time fraction for each candidate. There is no clear winner among the candidates. The MDD Bounding (Min. CD, desc, FS, RC) displays very impressive performance on several graphs, but does not finish on others. Among the variants that finished on all of the graphs, the MDD Bounding (Min. CD, asc, no FS, RC) is chosen as the representative algorithm, due to its high performance on the largest graph (HR(20)) and its reasonably good performance on all other graphs.

Table 4.30: Maximum times of Framework 3.1 variants on Hex Rook graphs.

Algorithm	**Time (seconds)**							**Avg. Frac.**
	HR (14)	HR (15)	HR (16)	HR (17)	HR (18)	HR (19)	HR (20)	
DD Bounding (Min. CD, asc, FS, no RC)	2.695	2.875	55.23	62.3	1283	1427	5621	0.529
DD Bounding (Min. CD, asc, no FS, no RC)	2.705	2.874	55.25	62.34	1286	1435	5695	0.527
MDD Bounding (Min. CD, asc, FS, RC)	2.154	3.693	42.62	56.79	1542	1330	5067	0.566
MDD Bounding (Min. CD, asc, no FS, RC)	2.142	3.743	41.87	57.73	1419	1320	5078	0.579
MDD Bounding (Min. CD, asc, no FS, no RC)	2.159	3.702	41.65	56.71	1414	1334	5313	0.578
MDD Bounding (Min. CD, desc, FS, RC)	–	0.897	–	20.67	–	433.6	1145	0.571
MDD Bounding (Min. CD, desc, no FS, RC)	–	0.906	–	20.88	–	433	1160	0.567
SageMath	127.1	78.45	–	6438	–	–	–	0.004

Only one graph from each of the Kneser and Knight graph families was classified as moderately difficult. Tables 4.31 and 4.32 show the results of a comparison of candidates on these graphs. For the Kneser graphs, the candidates were selected to be algorithms whose maximum running time was at most 200% of the best maximum running time. For the lone moderately difficult Knight graph, SageMath was the clear best solver. To compare the variants of Framework 3.1, the candidates were selected to be algorithms whose maximum running time was at most 150% of the best maximum running time among variants of Framework 3.1.

The best Framework 3.1 variant for the Kneser $(9, 4)$ was MDD Bounding (Min. CD, desc, no FS, RC), with MDD Bounding (Min. CD, desc, FS, RC) also demonstrating good performance. Since the only difference between these two variants is the force-stop optimization, the time difference appears to be entirely due to overhead related to the optimization, so MDD Bounding (Min. CD, desc, no FS, RC) is chosen as the representative algorithm for this case. Since this choice is based on data for only one graph, further study is needed to identify a high quality solver for Kneser graphs.

The same situation occurred for Knight (11): MDD Bounding (Min. CD, desc, no FS, RC) displayed the best performance among Framework 3.1 variants, with MDD Bounding (Min. CD, desc, FS, RC) also performing well. Based on the limited data available, the MDD Bounding (Min. CD, desc, no FS, RC) variant is the clear choice for a representative algorithm among the framework algorithms, but is still eclipsed by SageMath.

Table 4.31: Maximum times of Framework 3.1 variants on Kneser graphs.

Algorithm	**Time (seconds)**
	Kneser $(9, 4)$
MDD Bounding (Min. CD, asc, FS, RC)	839.2
MDD Bounding (Min. CD, asc, FS, no RC)	836.6
MDD Bounding (Min. CD, asc, no FS, RC)	778.3
MDD Bounding (Min. CD, asc, no FS, no RC)	817.9
MDD Bounding (Min. CD, desc, FS, RC)	438
MDD Bounding (Min. CD, desc, FS, no RC)	576.2
MDD Bounding (Min. CD, desc, no FS, RC)	420.8
MDD Bounding (Min. CD, desc, no FS, no RC)	558.7
SageMath	5677

Table 4.32: Maximum times of Framework 3.1 variants on Knight graphs.

Algorithm	**Time (seconds)**
	Knight (11)
MDD Bounding (Min. CD, desc, FS, RC)	6.117
MDD Bounding (Min. CD, desc, FS, no RC)	8.144
MDD Bounding (Min. CD, desc, no FS, RC)	6
MDD Bounding (Min. CD, desc, no FS, no RC)	8.029
SageMath	0.713

Five graphs from the set of Cartesian products of cycles were moderately difficult. As with the Knight graphs, SageMath was the best solver for all five cases. None of the framework algorithms finished on $C_{14} \,\square\, C_{14}$ or $C_{15} \,\square\, C_{15}$. Table 4.33 therefore excludes these two graphs from the comparison (and since it is a foregone conclusion that SageMath is a better choice for the graph family, the goal of the comparison is to track which framework variant is most competitive). Algorithms whose maximum running time was within 500% of the best maximum running time of any variant of Framework 3.1 were chosen as candidates, yielding a set of nine candidates. Among the candidates, the MDD Bounding (Min. CD, desc, no FS, RC) variant displays the best performance in all cases, with very little serious competition from the other candidates, and is the only candidate to finish on all three graphs (although, as mentioned previously, it did not finish on two of the graphs in the family).

Table 4.33: Maximum times of Framework 3.1 variants on Cartesian Products of Cycles.

Algorithm	**Time (seconds)**			**Avg. Frac.**
	$C_{11} \square C_{11}$	$C_{12} \square C_{12}$	$C_{13} \square C_{13}$	
MDD Bounding (Min. CD, asc, FS, RC)	18.74	448.6	–	0.060
MDD Bounding (Min. CD, asc, FS, no RC)	18.18	440.3	–	0.061
MDD Bounding (Min. CD, asc, no FS, RC)	17.59	453.6	–	0.062
MDD Bounding (Min. CD, asc, no FS, no RC)	16.92	403.5	–	0.066
MDD Bounding (Min. CD, desc, FS, RC)	5.413	183.8	5807	0.213
MDD Bounding (Min. CD, desc, FS, no RC)	8.355	241.4	–	0.128
MDD Bounding (Min. CD, desc, no FS, RC)	4.821	180.6	5480	0.234
MDD Bounding (Min. CD, desc, no FS, no RC)	7.896	241.4	7176	0.151
MDD Bounding (Min. MDD, desc, no FS, RC)	23.45	1212	–	0.041
SageMath	2.501	20.46	378.4	1.000

Three queen graphs were classified as moderately difficult: Queen (13) , Queen (14) and Queen (15). Algorithms whose maximum running time were at most 150% of the best maximum running time on any of these graphs were chosen as candidates, yielding six variants. Among these, the MDD Bounding (Min. MDD, desc, no FS, RC) displays the best performance in all cases, with MDD Bounding (Min. MDD, desc, FS, RC) also displaying good performance. As with the Knight graphs and Kneser graphs, the tight correlation of these two variants appears to be related to the force-stop optimization having little impact on the computation besides contributing overhead. The representative for this family was therefore chosen to be the MDD Bounding (Min. MDD, desc, no FS, RC) variant. Although the queen graphs represent only a small part of the input dataset for the experiments in this chapter, the results for the moderately difficult queen graphs were very useful for choosing a solver to solve open cases of the queen problem (documented in Chapter 5).

Table 4.34: Maximum times of Framework 3.1 variants on Queen graphs.

Algorithm	**Time (seconds)**			**Avg. Frac.**
	Queen (13)	Queen (14)	Queen (15)	
DD Bounding (Min. CD, asc, FS, no RC)	10.27	177.1	4302	0.530
DD Bounding (Min. CD, asc, no FS, RC)	10.74	185.7	4456	0.509
DD Bounding (Min. CD, asc, no FS, no RC)	10.28	177.1	4277	0.532
MDD Bounding (Min. CD, asc, FS, RC)	10.11	184.4	4179	0.532
MDD Bounding (Min. MDD, desc, FS, RC)	3.468	96.74	3075	0.994
MDD Bounding (Min. MDD, desc, no FS, RC)	3.412	96.59	3069	1.000
SageMath	223.4	–	–	0.005

The four graphs TG (15) , TG (16) , TG (17) and TG (18) from the triangular grid graphs

were classified as moderately difficult. SageMath had the best performance on all of these graphs. To compare variants of Framework 3.1, all algorithms whose maximum running time was at most 150% of the best maximum running time among variants of the framework were chosen as candidates. Among these, the MDD Bounding (Min. MDD, desc, no FS, RC) variant displayed the best performance among framework algorithms in all cases, with the corresponding 'FS' variant close behind. However, the advantage of SageMath in all cases is so great that none of the Framework 3.1 variants are competitive.

Table 4.35: Maximum times of Framework 3.1 variants on Triangular Grid graphs.

Algorithm	**Time (seconds)**				**Avg.**
	TG (15)	TG (16)	TG (17)	TG (18)	**Frac.**
MDD Bounding (Min. CD, desc, FS, RC)	3.231	35.33	151	3260	0.005
MDD Bounding (Min. CD, desc, no FS, RC)	3.065	34.13	149.2	3141	0.005
MDD Bounding (Min. MDD, desc, FS, RC)	2.598	23.21	117.8	845.5	0.006
MDD Bounding (Min. MDD, desc, no FS, RC)	2.479	22.3	112.9	842.9	0.007
SageMath	0.051	0.095	0.159	0.554	1.000

Chapter 5

New Domination Results for Queen Problems

This chapter describes several new results on queen problems, including solutions to several open cases of the queen domination problem, queen independent domination problem and the border queen problem. A summary of the solved cases is presented in Table 5.1 below. The solver variant used for the queen graph cases was the 'MDD Bounding (Min. MDD, desc, no FS, RC)' variant, which demonstrated high performance on queen graphs in the experiments in Chapter 4. Several modifications were made to the basic algorithm for the computations in this chapter. Sections 5.1 and 5.2 describe these enhancements. In addition, Section 5.5 presents a new set of upper bounds on the border queen problem using a classification for symmetric border sets. The solved cases of the queen domination problem are significant advances for such a difficult problem.

Most of the results in this chapter build on previous results, which were summarized concisely in an article by Kearse and Gibbons [42]. The results in [42] span many of the domination parameters for queen graphs, and include reports of the number of dominating sets and independent dominating sets of various sizes up to isomorphism. The results

presented in this chapter follow a similar protocol.

Table 5.1 gives the minimum orders of dominating sets, independent dominating sets and border dominating sets for $10 \leq n \leq 24$. The new results are in red/bold. Existing results for the non-border problems are taken from [42], and results for the border problems are taken from Sinko and Slater [56].

Table 5.1: Domination Numbers of Queen Graphs

n	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\gamma(\text{Queen}\,(n))$	5	5	6	7	8	9	9	9	9	10	**11**	11	**12**	12	**13**
$i(\text{Queen}\,(n))$	5	5	7	7	8	9	9	9	10	**11**	**11**	11	**12**	**13**	**13**
$\text{bor}\,(\text{Queen}\,(n))$	6	9	10	9	**12**	**13**	**10**	**14**	**16**	**13**	**18**	**19**	**14**	**21**	**22**

5.1 Computing Independent Dominating Sets

Algorithms based on Framework 3.1 can be used to find $\gamma(\text{Queen}\,(n))$ directly, and also to find $\text{bor}\,(\text{Queen}\,(n))$ by restricting the initial candidate set C to contain only vertices on the border of the board. To compute $i(\text{Queen}\,(n))$, however, additional logic is needed. One option for using Framework 3.1 to generate independent dominating sets is to generate all dominating sets of the desired size and discard all sets which are not independent (similar to the method described later in this Chapter for generating rotated border constructions for the border queen problem). However, for independent dominating sets, a small addition to the logic in Framework 3.1 can ensure that the only recursive branches followed are those which may produce an independent set.

Consider the following snippet of pseudocode, which is excerpted from lines 11 - 16 of Framework 3.1.

$T \leftarrow \emptyset$
$v \leftarrow$ An undominated vertex of G
for each vertex $u \in N[v] \cap C$ **do**
    $T \leftarrow T \cup \{u\}$

FindDominatingSet($G, P \cup \{u\}, C - T, B,$ `desired_size`)
end for

The candidate set provided in the recursive call to FindDominatingSet is $C-T$, where C is the candidate set provided to the current invocation of FindDominatingSet and T is a set containing all vertices which have been used as dominators in previous iterations of the loop. Only vertices in the set $C-T$ will be used as dominators in future branches. To ensure that the resulting dominating set is an independent set, it suffices to ensure that no vertex in the candidate set is ever adjacent to a vertex in the under-construction dominating set P. Therefore, algorithms based on Framework 3.1 can be modified to generate independent dominating sets by changing the $C - T$ term in the recursive call to $C - T - N[u]$.

5.2 Splitting Computation Among Processes

To leverage the extra computational power of multi-processor machines, the recursive backtracking search can be split among multiple processes by partitioning the search tree. For an algorithm based on Framework 3.1, define the *recursive depth* of a particular call to FindDominatingSet to be the size of the working set P when the FindDominatingSet function begins. Define the *search tree* to be a rooted tree T containing a node for each call to FindDominatingSet over the entire recursive search, with the parent of each node being either the FindDominatingSet call which created the node. The initial call (which starts the recursive process) has no parent and is the root of the tree. Note that since each recursive call to FindDominatingSet (besides the initial call which starts recursion) is made after adding a vertex to P, the number of elements in P in a call to FindDominatingSet is always equal to the depth of the corresponding node in T.

To split a single recursive search over $k \geq 2$ processes, the search tree is partitioned among the k processes such that the overall set of nodes visited by the search is identical (although some nodes may be visited by more than one process). Ideally, the partitioning

strategy should ensure that the number of duplicated nodes is minimized and that each process receives an equal portion of the tree. Figure 5.1 shows two prospective strategies for splitting the computation tree among 3 processes. In both diagrams, uncoloured nodes are computed by all processes and coloured nodes are computed by a particular process only. In Figure 5.1a, the tree is divided at the root into three contiguous subtrees, each of which is computed by a different process. This partitioning strategy is reasonably straightforward for implementations of Framework 3.1: the subtrees for each process can be determined by the initial choice of vertex to add to the set. Additionally, splitting the computation at the highest possible point in the tree has the advantage of minimizing the number of uncoloured nodes, which must be computed separately by all processes. However, the strategy in Figure 5.1a lacks granularity. For example, suppose that the tree pictured in Figure 5.1a continues beyond level 3. The structure of the tree is not known in advance, so there is a possibility that most of the processors receive a very small subtree and one processor receives a huge subtree that contains the majority of the nodes in the computation. This will result in a very unbalanced partition of nodes among processes.

An alternative method is illustrated in Figure 5.1b. This method assigns a larger number of subtrees among processes, with the set of subtrees given to each process sampled from different branches of the full computation tree. Specifically, a starting depth is chosen, and the nodes at that depth are divided among the different processors. For each depth d, let $N_{d,0}, N_{d,1}, \ldots, N_{d,k}$ be the set of nodes of the tree at depth d from the root (which is the initial call to FindDominatingSet), with the order of nodes corresponding to the order in which they are traversed by the backtracking algorithm (since the algorithm is deterministic and all processes compute the same set of nodes in the uncoloured part of the tree, this ordering will be fixed and consistent for a particular graph and algorithm). If there are q processes $P_0, P_1, \ldots, P_{q-1}$, then the computation can be split among the processes at depth d by assigning the subtrees of all nodes $N_{d,i}$ with $i \equiv j \pmod{q}$ to process P_j for $0 \leq i \leq k-1$.

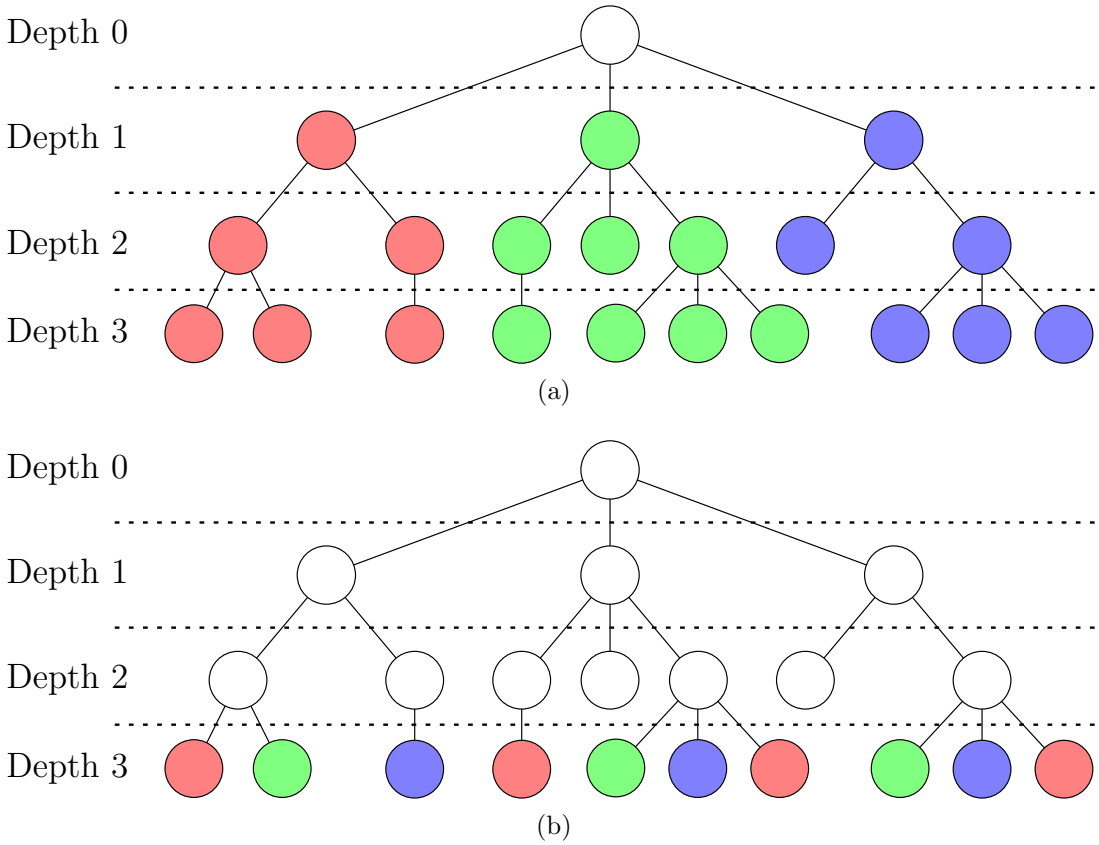


Figure 5.1: Two strategies for splitting a computation among multiple processes (indicated by different colours).

This method requires more duplication of nodes among all processes, since for a particular process to iterate over its assigned subtrees, it must compute all of the upper levels of the tree. Since this multiprocessing strategy divides nodes among processes by arithmetic modulo the number of processes q and identifying each process with a particular residue modulo q, it is referred to in the remainder of this document (namely Chapter 6) as the *res/mod approach*. To solve the open cases of the queen domination problems, the res/mod approach was used to split the computation among a cluster of 64 processors. The splitting depth was chosen by hand for each computation.

5.3 Counting Solutions up to Isomorphism

As mentioned in the introduction to this chapter, the most complete recent article regarding solved cases of the queen domination problem is by Kearse and Gibbons [42] and presents solutions to $\gamma(\text{Queen}\,(n))$ for $n \leq 19$. The data produced by [42] also includes the number of minimum dominating sets of queen graphs up to isomorphism for some of the solved cases. Data for $n \leq 11$ was produced earlier by Burger [12]. This data can be useful for reproducing the results and verifying the correctness of an implementation of a domination solver. Tables 5.2, 5.3 and 5.4 summarize (respectively) the number of minimum dominating sets, minimum independent dominating sets and minimum border dominating sets up to isomorphism. It was not possible to count the total number of solutions up to isomorphism for all of the solved cases (since exhaustively generating all sets up to isomorphism requires significantly longer than solving the optimization problem), but the data in Tables 5.2, 5.3 and 5.4 contains cases not covered by [42] and matches the data in [42] for all cases covered by that article. This, combined with the fact that the solver implementation yielded correct results for all of the experiments in Chapter 4, seems to indicate that the solver implementation was correct.

The new results in Table 5.2 are shown in red. None of the results in Table 5.3 are new, and all of the counts in Table 5.4 are new (since previous research on border domination has not covered the generation of all sets up to isomorphism).

Table 5.2: Number of Minimum Dominating Sets of Queen Graphs up to Isomorphism

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Size	1	2	3	3	4	5	5	5	5	6	7	8	9	9	9	9
#	1	3	37	1	13	638	21	1	1	1	41	588	25872	**43**	**22**	**2**

Table 5.3: Number of Minimum Independent Dominating Sets of Queen Graphs up to Isomorphism

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Size	1	3	3	4	4	5	5	5	5	7	7	8	9	9	9	10
#	1	2	2	17	1	91	16	1	1	105	4	55	1314	16	2	28

Table 5.4: Number of Minimum Border Dominating Sets of Queen Graphs up to Isomorphism

n	3	4	5	6	7	8	9	10	11	12
Size	2	2	3	4	5	6	6	6	9	10
#	4	1	6	19	75	174	1	1	1017	979
n	13	14	15	16	17	18	19	20	21	22
Size	9	12	13	10	14	16	13	18	19	14
#	4	1094	2635	2	32	1457	8	2080	6128	4

5.4 Certificates of Independent Dominating Sets

Figures 5.2, 5.3, 5.4, 5.5, and 5.6 show minimum independent dominating sets of Queen (19), Queen (20), Queen (22), Queen (23), and Queen (24), respectively. For $n = 20, 22$ and 24, these are also minimum dominating sets. See Section 5.5.2 for a discussion of the new border queen results.

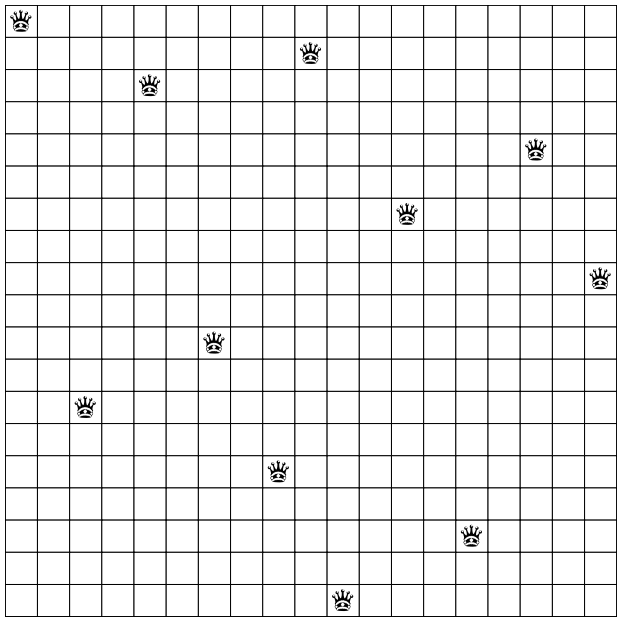


Figure 5.2: An independent dominating set of Queen (19) with size 11.

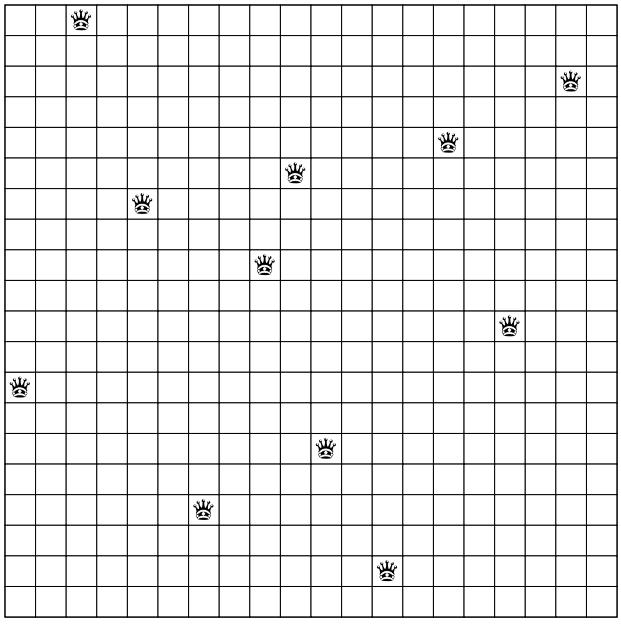


Figure 5.3: An independent dominating set of Queen (20) with size 11.

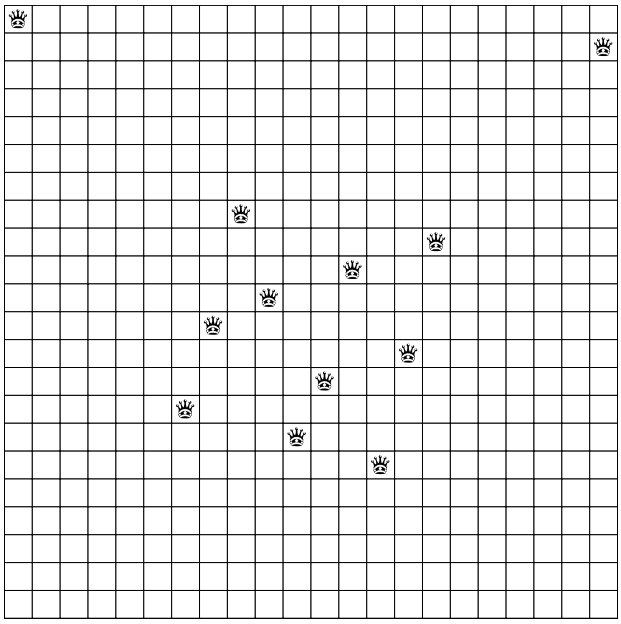


Figure 5.4: An independent dominating set of Queen (22) with size 12.

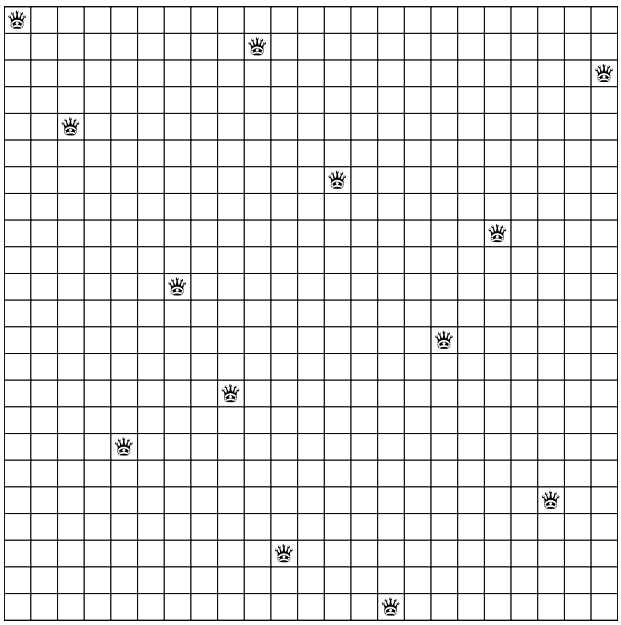


Figure 5.5: An independent dominating set of Queen (23) with size 13.

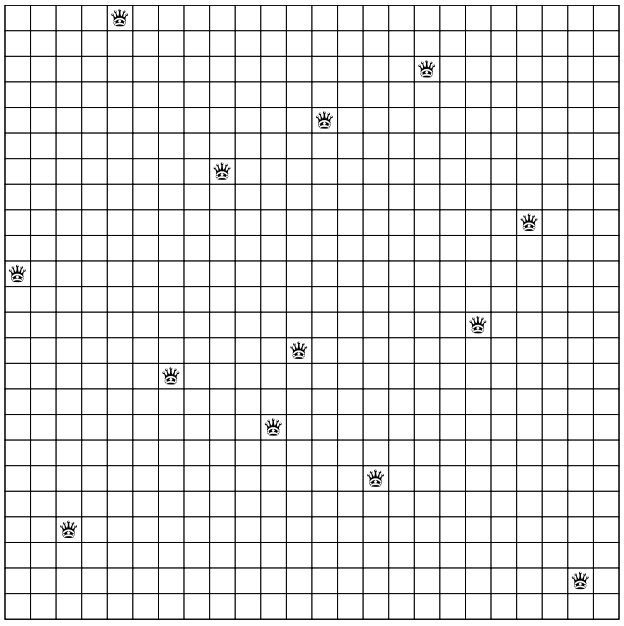


Figure 5.6: An independent dominating set of Queen (24) with size 13.

5.5 Rotated Border Constructions

This section presents several constructions for border dominating sets of Queen (n) which use rotational symmetry. Two of the constructions were proved by Sinko and Slater in their original paper [56], and the remainder are new to this work. The formal constraints of the symmetric sets are given in Definition 5.1.

Definition 5.1. *A* rotated border construction *(RBC) is defined to be a border dominating set S of Queen (n) (with $n \geq 3$) such that if $v_{i,j}$ is in S for $0 \leq i, j \leq n-1$, then exactly one of the following conditions applies.*

1. *$v_{i,j}$ is a* corner vertex*: $(i, j) \in \{0, n-1\} \times \{0, n-1\}$.*

2. *$v_{i,j}$ is a* midpoint vertex*: n is odd and $v_{i,j}$ lies at the midpoint of a border segment.*

Formally,

$$(i, j) \in \{(0, (n-1)/2), (n-1, (n-1)/2), ((n-1)/2, 0), ((n-1)/2, n-1)\}$$

3. *If $v_{i,j}$ is neither a corner or midpoint vertex, then all of the images of $v_{i,j}$ under rotational symmetries of the board must also be in S. Specifically, each of the following vertices must be in S:*

$$\begin{aligned} 0^\circ \textit{ rotation:} &\quad v_{i,j} \\ 90^\circ \textit{ rotation:} &\quad v_{j,(n-1)-i} \\ 180^\circ \textit{ rotation:} &\quad v_{(n-1)-i,(n-1)-j} \\ 270^\circ \textit{ rotation:} &\quad v_{(n-1)-j,i} \end{aligned}$$

Vertices falling into this case are called rotated vertices.

The three conditions of Definition 5.1 allow an RBC to be specified by providing the status of each corner vertex, each midpoint vertex (if n is odd) and the status of each rotated vertex in row 0 (since the remaining vertices can be obtained by applying all rotations to the row 0 vertices). An RBC S over Queen (n) is *minimal* if there no proper subset $T \subset S$ is also an RBC. An RBC S over Queen (n) is *minimum* if, for all other RBCs T over Queen (n), $|T| \geq |S|$. Denote by MinRBC (n) the minimum size of an RBC over Queen (n). One of the culminating results of this section, Theorem 5.22, establishes that MinRBC (n) exists and is at most n for all n. Note that while a minimal RBC must by definition be a border dominating set, it is not necessarily a minimal border dominating set. An RBC is said to be *degenerate* if it consists entirely of corner and midpoint vertices and contains no rotated vertices. Lemmas 5.2 and 5.3 establish that all RBCs of sufficiently large boards are non-degenerate. The asymmetry between the even and odd cases is caused by the special treatment afforded to midpoint vertices in the definition above (which, in turn, is guided by the observations made regarding redundancy in Lemmas 5.4 and 5.6 later in this section).

Lemma 5.2. *For even $n \geq 6$, no degenerate RBCs exist.*

Proof. Since no midpoint vertices exist for even n, a degenerate RBC can only contain corner vertices, which can only dominate border vertices and vertices on the forward- or back-diagonal. For all $n \geq 6$, Queen (n) contains interior vertices which are not on either diagonal, and therefore cannot be dominated by a degenerate RBC. □

Lemma 5.3. *For odd $n \geq 9$, no degenerate RBCs exist.*

Proof. For each odd $n \leq 7$, a degenerate RBC exists (for example, by taking all corner and midpoint vertices). For $n \geq 17$, Theorem 2.1 (which establishes that any dominating set of Queen (n) for odd $n \geq 13$ must have size at least $(n+1)/2)$ prevents the existence of a degenerate RBC since the maximum size of an RBC is eight. Let $n \geq 9$ and consider the set S of all 8 corner and midpoint vertices. Any degenerate RBC must be a subset of S. We will show that vertex $v_{1,2}$ is not dominated by any vertex in S.

The corner vertices cannot dominate $v_{1,2}$, because it is neither a border vertex nor on the forward- or back-diagonal of the board. Since $n \geq 9$ implies that $(n-1)/2 \geq 4$, $v_{1,2}$ cannot lie in the same row or column as a midpoint vertex. Additionally, $v_{1,2}$ does not meet Conditions 2.5 or 2.6 to have any of the four midpoint vertices as a neighbour:

- $v_{0,(n-1)/2} : 0 - 1 \neq (n-1)/2 - 2$ and $0 - 1 \neq 2 - (n-1)/2$ since $(n-1)/2 \geq 4$.
- $v_{n-1,(n-1)/2} : (n-1) - 1 \neq (n-1)/2 - 2$ and $(n-1) - 1 \geq 2 - (n-1)/2$.
- $v_{(n-1)/2,0} : (n-1)/2 - 1 \neq 0 - 2$ and $(n-1)/2 - 1 \neq 2 - 0$ since $(n-1)/2 \geq 4$.
- $v_{(n-1)/2,n-1} : (n-1)/2 - 1 \neq (n-1) - 2$ and $(n-1)/2 \neq 2 - (n-1)$

□

Lemmas 5.2 and 5.3 imply that all RBCs for Queen (n) with $n \geq 9$ must contain at least one rotated vertex, with the effect that each of the four border segments will contain at

least one vertex. As a result, no corner or midpoint vertices are needed to cover the border vertices, so the vertices uniquely dominated by corner or midpoint vertices must be interior. Lemmas 5.4 and 5.6 use this fact to restrict the number of corner and midpoint vertices needed for a minimal RBC. Additionally, Lemma 5.7 establishes that if only two midpoints are needed, it is sufficient to use the midpoint of row 0 and the midpoint of column 0.

Lemma 5.4. *Let S be an RBC of Queen(n) which contains two diagonally opposite corners. If S is non-degenerate, then S is not minimal.*

Proof. Suppose S is non-degenerate and contains two diagonally-opposite corners, and without loss of generality, assume that the two corners are $v_{0,0}$ and $v_{n-1,n-1}$. Since S is non-degenerate, it contains at least one rotated vertex, whose four rotations lie on the four border segments and dominate all vertices on the border. The only other vertices in the neighbourhoods of $v_{0,0}$ and $v_{n-1,n-1}$ are on the forward diagonal, and both of the corner vertices dominate the diagonal. Therefore, the set $T = S - \{v_{0,0}\}$ is also an RBC, so S is not minimal. □

Corollary 5.5. *For $n \geq 9$, any minimal RBC on Queen(n) contains at most two corner vertices. Moreover, any minimal RBC is a rotation of a minimal RBC whose only corner vertices lie in row 0.*

Lemma 5.6. *Let S be an RBC of Queen(n) for odd n. If S is non-degenerate and contains all four midpoint vertices, then S is not minimal.*

Proof. Suppose S is non-degenerate and contains all four midpoint vertices. Consider the midpoint $v_{0,(n-1)/2}$ in row 0. Since S is non-degenerate, the four rotations of a rotated vertex together cover all border vertices. Column $(n-1)/2$ is dominated by both $v_{n-1,(n-1)/2}$ and $v_{0,(n-1)/2}$. Additionally, since $(n-1)/2 - 0 = 0 - (n-1)/2$, $v_{0,(n-1)/2}$ lies on the forward diagonal of the left midpoint $v_{(n-1)/2,0}$, and since $(n-1)/2 - 0 = (n-1) - (n-1)/2$, $v_{0,(n-1)/2}$ lies on the back diagonal of the right midpoint $v_{(n-1)/2,n-1}$. Therefore, all vertices on the

forward or back diagonal of $v_{0,(n-1)/2}$ are dominated by other midpoints, so $v_{0,(n-1)/2}$ does not uniquely dominate any vertices and the set $T = S - \{v_{0,(n-1)/2}\}$ is also an RBC. □

Lemma 5.7. *Let S be a minimal, non-degenerate RBC of Queen(n) for odd n, and suppose that S contains two midpoint vertices m_1 and m_2. Then m_1 and m_2 do not share a row or a column.*

Proof. Suppose m_1 and m_2 do share a row or column, and without loss of generality (by rotation), assume that the two midpoints are $v_{0,(n-1)/2}$ and $v_{n-1,(n-1)/2}$. We will establish that either S is not minimal or S is not a dominating set.

Since both $v_{0,(n-1)/2}$ and $v_{n-1,(n-1)/2}$ are present, there is at least one vertex which is diagonally dominated by a midpoint vertex and not dominated by any other vertices (since otherwise, one midpoint would suffice). Let $v_{i,j}$ be that vertex, and without loss of generality (by reflecting the board if necessary) assume that $v_{i,j}$ lies in the quadrant $0 \leq i, j < (n-1)/2$ and therefore that its sole dominator in S is $v_{0,(n-1)/2}$. Since $v_{i,j}$ is a back-diagonal neighbour of $v_{0,(n-1)/2}$, $i + j = (n-1)/2$. Since $v_{i,j}$ is not dominated by any vertices besides $v_{0,(n-1)/2}$, there cannot be a queen in row i or column j. Consider the vertex $v_{(n-1)/2,j}$, which lies in the middle row of the board. Since both midpoint vertices lie in column $(n-1)/2$, vertex $v_{(n-1)/2,j}$ cannot be row-dominated, and since there is no queen in column j, vertex $v_{(n-1)/2,j}$ cannot be column-dominated. Therefore, one of the four border vertices which are diagonal neighbours of $v_{(n-1)/2,j}$ is in S. Those neighbours are

$$v_{0,j+(n-1)/2} = 90^\circ \text{ rotation of } v_{(n-1)/2-j,0} = v_{i,0}$$

$$v_{n-1,j+(n-1)/2} = 90^\circ \text{ rotation of } v_{(n-1)/2-j,n-1} = v_{i,n-1}$$

$$v_{(n-1)/2-j,0} = v_{i,0} \text{ since } i + j = (n-1)/2$$

$$v_{(n-1)/2+j,0} = v_{(n-1)-j,0} = 270^\circ \text{ rotation of } v_{0,j}.$$

Each of the four vertices, if present in S, would be a rotated vertex, so all of its rotations

would also be in S. In all four cases, one of the rotations would place a queen in either row i or column j, which is a contradiction.

Therefore, if any vertex $v_{i,j}$ meeting the conditions above exists, S is not a dominating set. If no such $v_{i,j}$ exists, then one of the midpoint vertices in S is redundant and S is not minimal. □

Lemma 5.8 gives a condition for rotated vertices to be redundant, when their four rotations are considered.

Lemma 5.8. *Let S be an RBC of Queen(n), and let $v_{0,j} \in S$ be a rotated vertex. If $v_{0,(n-1)-j} \in S$, then S is not minimal. Specifically, all four rotations of $v_{0,(n-1)-j}$ are redundant.*

Proof. The four rotations of $v_{0,j}$ are

$$v_{0,j}, \quad v_{j,n-1}, \quad v_{n-1,(n-1)-j}, \quad v_{(n-1)-j,0}.$$

The vertex $v_{0,(n-1)-j}$ is the image of $v_{0,j}$ under a horizontal reflection, and each of the rotations of $v_{0,(n-1)-j}$ is the image of a rotation of $v_{0,j}$ under a horizontal or vertical reflection. We will show that every vertex dominated by $v_{0,(n-1)-j}$ is also dominated by a rotation of $v_{0,j}$. By rotational symmetry, this is sufficient to establish that every rotation of $v_{0,(n-1)-j}$ is redundant and S is not minimal.

Vertices in row 0 are clearly dominated by $v_{0,j}$. Similarly, vertices in column $(n-1)-j$ are dominated by the 180° rotation of $v_{0,j}$, namely $v_{(n-1),(n-1)-j}$.

The vertex $v_{0,(n-1)-j}$ is a diagonal neighbour of the 90° rotation of $v_{0,j}$, namely $v_{j,(n-1)}$, since (by Condition 2.5)

$$0 - j = ((n-1) - j) - (n-1),$$

and vertex $v_{0,(n-1)-j}$ is a back-diagonal neighbour of the 270° rotation of $v_{0,j}$, namely

$v_{(n-1)-j,0}$, since (by Condition 2.6)

$$0 - ((n-1) - j) = 0 - ((n-1) - j).$$

Therefore, all diagonal neighbours of $v_{0,(n-1)-j}$ are dominated by rotations of $v_{0,j}$. Since every neighbour of $v_{0,(n-1)-j}$ is dominated by a rotation of $v_{0,j}$, the vertex $v_{0,(n-1)-j}$ is redundant and S is not minimal. □

Corollary 5.9. *If $v_{i,j}$ is any rotation of a rotated vertex of an RBC S of Queen(n), then every vertex which neighbours any reflection of $v_{i,j}$ (horizontal, vertical, diagonal or back-diagonal) is dominated by S.*

Lemma 5.8 implies that if a rotated vertex $v_{0,j}$ is a member of an RBC S, a new RBC S' can be constructed by replacing the four rotations of $v_{0,j}$ with the four rotations of $v_{0,(n-1)-j}$. As a result, a minimal RBC containing a rotated vertex $v_{0,j}$ for $j > n/2$ corresponds to a minimal RBC with $v_{0,(n-1)-j}$ and its rotations replacing the rotations of $v_{0,j}$. This observation allows a compact and convenient classification for RBCs, which is formalized in Definition 5.10. Theorem 5.11 gives conditions for a canonical RBC to exist.

Definition 5.10. *A* canonical RBC *over Queen(n) is a non-degenerate RBC S in which each of the following conditions is met.*

1. *All rotated vertices $v_{0,j} \in S$ have $j < n/2$.*
2. *Any corner vertices in S are in row 0, and if S contains only one corner vertex, then it must be $v_{0,0}$.*
3. *If n is odd, S contains at most 3 midpoint vertices. Specifically if S contains one midpoint vertex, it must be $v_{0,(n-1)/2}$, if S contains two midpoint vertices, they must be $v_{0,(n-1)/2}$ and $v_{(n-1)/2,0}$, and if S contains 3 midpoint vertices, they must be $v_{0,(n-1)/2}$, $v_{(n-1)/2,0}$ and $v_{n-1,(n-1)/2}$*

Theorem 5.11. *Any minimal, non-degenerate RBC S has a canonical form.*

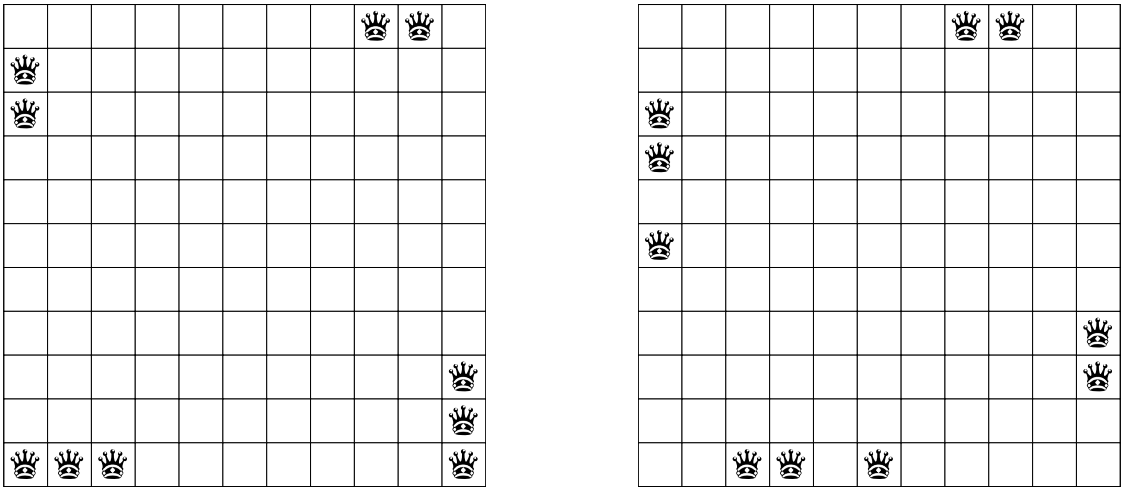
Proof. By Lemma 5.8, any rotated vertex can be replaced by its horizontal reflection. Therefore, S can be brought into compliance with Condition 1 in Definition 5.10 by replacing all rotated vertices in the right half of row 0 with their reflections in the left half.

By Lemma 5.4, no more than two corner vertices are present in any minimal, non-degenerate RBC, and, in general, the exact positions of corner vertices are irrelevant, so S can be brought into compliance with Condition 2 in Definition 5.10 by only using corner vertices from row 0.

Lemma 5.6 proves that no more than three midpoints will be present in a minimal, non-degenerate RBC. By rotational symmetry, if only one midpoint is needed, it can be assumed to be in row 0. Lemma 5.7 proves that if 2 midpoints are needed, they will not share a row or column. Together, these conditions are sufficient to enforce the particular ordering required by Condition 3 of Definition 5.10. □

The restrictions in 5.10 allow an RBC to be completely specified by giving the number of corner vertices (either 0, 1 or 2), the number of midpoint vertices (if n is odd, between 0 and 3, otherwise 0) and the rotated vertices in row 0 (which, by the definition, must all lie to the left of the midpoint of row 0. Therefore, we propose a compact notation for RBCs consisting of the first $\lceil n/2 \rceil$ cells of row 0, with queens in rotated cells denoted by Q and queens in corner (or midpoint) cells denoted by C (or M), superscripted by the number of corners (or midpoints) used.

In this section, a pattern specified by this notation will be generically called a *rotated pattern*, since it may not always be correct to call it an RBC until it is proven to be a dominating set. Although the rotated pattern notation only depicts the first half of row 0, any references to the set of vertices corresponding to a rotated pattern is assumed to refer to the set of vertices corresponding to all rotations of the pattern (including the 4 rotations of all rotated vertices and the specified multiplicities of the corner and midpoint vertices).



(a) (b)

Figure 5.7: Examples of two canonical RBCs on $n = 11$.

The RBC corresponding to the rotated pattern

C^2	Q	Q			

is shown in Figure 5.7a, and the RBC corresponding to the rotated pattern

		Q	Q		M^2

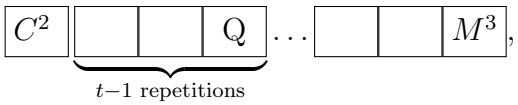
is shown in Figure 5.7b. Note that bor (Queen (11)) = 9, so the two RBCs shown in Figure 5.7 are not minimum border dominating sets.

The original paper on border domination by Sinko and Slater [56] included a theorem which constructively proved two bounds on bor (Queen (n)). The constructions can be represented by canonical RBCs, and are re-stated in Theorems 5.12 and 5.13 below (the original proof in [56] combines both into a single statement, but since the constructions have notable differences, they are presented separately here). The theorems are rephrased for stylistic consistency with the other theorems in this section, and the proofs have been adapted to use

the RBC notation introduced here.

Theorem 5.12 (Sinko and Slater [56])**.** *If* $n \equiv 1 \pmod 6$*, then* $bor(Queen(n)) \leq \frac{2(n-1)}{3} + 1$.

Proof. Let $n \equiv 1 \pmod 6$. If $n = 1$, then the statement holds since $\text{bor}(\text{Queen}(1)) = 1$. Otherwise, assume $n \geq 7$ and represent n as $n = 6t + 1$, where $t \geq 1$ by the assumption above. Note that there are $(n + 1)/2 = 3t + 1$ cells between the lower left corner and the midpoint of row 0 (inclusive). We will show that the rotated pattern



which contains $2 + 4(t - 1) + 3 = 4t + 1 = \frac{2(n-1)}{3} + 1$ cells, is an RBC.

Let S be the set of vertices corresponding to the rotated pattern above.

The four rotations of the pattern place a queen in every column c where $c \equiv 0 \pmod 3$ and every row r such that $r \equiv 0 \pmod 3$. Therefore, every vertex $v_{i,j}$ where either i or j is a multiple of 3 is dominated by a vertex in its row or column. Since 0 and $n - 1$ are both multiples of 3, all border vertices are dominated.

The two cases below demonstrate that all vertices $v_{i,j}$ below the main diagonal (that is, with $i \leq j$) which do not fall into the cases above are dominated by a vertex in S.

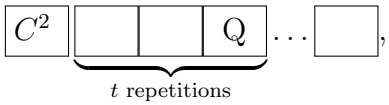
1. $i \equiv j \pmod 3$: Consider the vertex $v_{0,j-i}$, which is a forward-diagonal neighbour of $v_{i,j}$. Since $j - i \equiv 0 \pmod 3$, either $v_{0,j-i}$ is in S or $v_{0,j-i}$ is the horizontal reflection of a vertex in S. In the former case, $v_{i,j}$ is dominated by $v_{0,j-i}$. In the latter case, by Corollary 5.9, $v_{i,j}$ is dominated by a reflection of $v_{0,j-i}$.

2. $i \equiv -j \pmod 3$: Consider the vertex $v_{0,j+i}$, which is a back-diagonal neighbour of $v_{i,j}$. Since $j + i \equiv 0$, either $v_{0,j+i}$ is in S or $v_{0,j+i}$ is the horizontal reflection of a vertex in S, and as above, this implies that $v_{i,j}$ is dominated.

By 180° rotational symmetry, since every vertex on or below the main diagonal is dominated, S is a dominating set, so the rotated pattern above is an RBC. □

Theorem 5.13 (Sinko and Slater [56])**.** *If* $n \equiv 4 \pmod 6$*, then* $bor(Queen(n)) \leq \frac{2(n-1)}{3}$*.*

Proof. Let $n \equiv 4 \pmod 6$ and represent n as $n = 6t + 4$. We will show that the rotated pattern



which contains $2 + 4t = 2 + 4(n-4)/6 = \frac{2(n-1)}{3}$ cells, is an RBC.

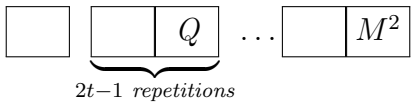
The rightmost rotated queen on row 0 is in column $3t = (n-4)/2$. The 180° rotation of $v_{0,(n-4)/2}$ is in column $(n+2)/2 = (n-4)/2 + 3$, so the rotated pattern places a queen in every third column, including columns 0 and $n-1$ which contain corner vertices. By rotation, a queen is also placed in every third row. The same argument used in the proof of Theorem 5.12 then applies to demonstrate that every cell on the board is dominated by at least one vertex in the rotated pattern, so the pattern is an RBC. □

Theorem 5.13, together with the bound given by Theorem 2.11, imply a solution to the border queen problem for $n \equiv 4 \pmod 6$.

Corollary 5.14. *If* $n \equiv 4 \pmod 6$*, then* $bor(Queen(n)) = 2(n-1)/3$*.*

Theorems 5.15 - 5.19 give constructions for RBCs for all $n \not\equiv 7 \pmod 8$. Theorem 5.15 also improves on the upper bound of $n-2$ for cases where $n \equiv 1 \pmod 8$.

Theorem 5.15. *If* $n = 8t + 1$ *for* $t \geq 1$*, then the rotated pattern*



is an RBC of size $2 + 4(2t-1) = 8t - 2 = n - 3$*.*

Proof. Let S be the vertex set of Queen (n) produced by the rotated pattern above. First, observe that every even numbered row or column contains a queen (including all of the border rows and columns, even though corner vertices are not used in the pattern. It is therefore sufficient to prove that all $v_{i,j}$, with i and j odd, are dominated. The only parts of the pattern that are not completely symmetric about rotations are the midpoint cells, which ensure that the midpoint rows and columns (which are even-numbered) are dominated. Cells with even row or column numbers are all dominated by the rotated vertices, and by rotational symmetry, it is sufficient to prove that every $v_{i,j}$, with both $i, j \equiv 1 \pmod 3$ and $i, j < (n-1)/2$, is dominated by a rotated vertex.

Consider a vertex $v_{i,j}$ meeting the conditions above.

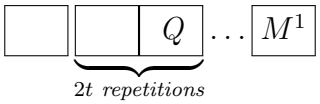
1. If $i+j \neq (n-1)/2$, then consider the vertex $v_{0,j+i}$, which is a back-diagonal neighbour of $v_{i,j}$. Column $j+i$ is even, and by the condition $i, j < (n-1)/2$, vertex $v_{0,j+i}$ is not a corner vertex. Since $i+j \neq (n-1)/2$, vertex $v_{0,j+i}$ is also not a midpoint vertex (since the only back-diagonal neighbours of the midpoint $v_{0,(n-1)/2}$ have $i+j \neq (n-1)/2$. Therefore, either $v_{0,j+1}$ is a rotated vertex or its horizontal reflection $v_{0,(n-1)-(j+i)}$ is a rotated vertex. By Corollary 5.9, both cases imply that $v_{i,j}$ is dominated.

2. If $i+j \neq (n-1)/2$, then as noted above the vertex $v_{0,j+i}$ is a midpoint vertex. However, consider the forward diagonal neighbour $v_{n-1,j+(n-1)-i} = v_{n-1,(n-1)-(i-j)}$. Since $n-1$ and $i-j$ are both even, the vertex $v_{n-1,(n-1)-(i-j)}$ lies in an even column on the top border row. Since $1 \leq i, j < (n-1)/2$, column $(n-1)-(i-j)$ cannot be a border column or a midpoint column. Therefore, either $v_{n-1,(n-1)-(i-j)}$ is a rotated vertex or a reflection of a rotated vertex, so Corollary 5.9 implies that $v_{i,j}$ is dominated.

Therefore, S is a dominating set.

□

Theorem 5.16. *If $n = 8t + 3$ for $t \geq 1$, then the rotated pattern*



is an RBC of size $8t + 1 = n - 2$.

Proof. Let S be the vertex set of Queen (n) produced by the rotated pattern above. As with Theorems 5.15 and 5.17, every even numbered row and column contains a queen. Let $v_{i,j}$ be a vertex with both i and j odd, $i \leq j$ and $i \leq (n-1) - j$.

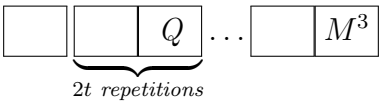
1. If $i = j$ and $i + j < n - 1$, then the back-diagonal neighbour $v_{0,j+i}$ is in S, since $i + j$ is even and $v_{0,j+i}$ is not a corner vertex.

2. If $i = j = (n-1)/2$, then the back-diagonal neighbour $v_{0,j+i}$ is not in S, but the single midpoint vertex dominates $v_{i,j}$.

3. If $i \neq j$, then the forward-diagonal neighbour $v_{0,j-i}$ is not a corner vertex, and is in S since $j - i$ is even.

Therefore, S is a dominating set. □

Theorem 5.17. *If $n = 8t + 5$ for $t \geq 1$, then the rotated pattern*



is an RBC of size $8t + 2 = n - 2$.

Proof. Let S be the vertex set of Queen (n) produced by the rotated pattern above. As in the proof of Theorem 5.15, observe that every even numbered row and column contains a queen. Since three midpoint vertices are present, Lemma 5.6 allows the assumption that all four midpoint vertices are present, so the four border segments are perfectly symmetric.

Consider a vertex $v_{i,j}$ where both i and j are odd, $i \leq j$ and $i \leq (n-1) - j$. By symmetry, if every such vertex $v_{i,j}$ is covered, then every rotation of such a vertex is also covered.

1. If $i = j$, then the back-diagonal neighbour $v_{0,j+i}$ is in S, since $i + j$ is even and $v_{0,j+i}$ is not a corner vertex.

2. If $i \neq j$, then the forward-diagonal neighbour $v_{0,j-i}$ is not a corner vertex, and is in S since $j - i$ is even.

Therefore, S is a dominating set. □

Theorem 5.18. *If $n = 4t$ for $t \geq 1$, then the rotated pattern*

$$\boxed{C^2}\ \underbrace{\boxed{Q}\ldots}_{t-1 \text{ repetitions}}$$

is an RBC of size $4(t-1) + 2 = n - 2$.

Proof. Let S be the vertex set of Queen (n) produced by the rotated pattern above. Since both corner vertices are included, by Lemma 5.4 it is possible to assume that the pattern is perfectly symmetric under rotation (that is, that all four corners are present). Rows and columns with indices in $\{0, 1, \ldots, t-1, 3t, 3t+1, \ldots, 4t-1\}$ contain a queen. Let $v_{i,j}$ be a vertex with $i \leq j$, $i \leq (n-1) - j$ and $i, j \in \{t, t+1, \ldots, 3t-1\}$.

1. If $j - i < t$, then the forward-diagonal neighbour $v_{0,j-i}$ of $v_{i,j}$ is in S.

2. If $j - i \geq t$, then $j + i = 2i + (j - i) \geq 3t$, so the back-diagonal neighbour $v_{0,j+i}$ is in S.

Therefore, S is a dominating set. □

Theorem 5.19. *If $n = 4t + 2$ for $t \geq 1$, then the rotated pattern*

$$\boxed{C^2}\ \underbrace{\boxed{Q}\ldots}_{t \text{ repetitions}}$$

is an RBC of size $4t + 2 = n$.

Proof. Let S be the vertex set of Queen (n) produced by the rotated pattern above. Since both corner vertices are included, by Lemma 5.4 it is possible to assume that the pattern is perfectly symmetric under rotation (that is, that all four corners are present). Rows and columns with indices in $\{0, 1, \ldots, t, 3t + 1, 3t + 2, \ldots, 4t + 1\}$ contain a queen. Let $v_{i,j}$ be a vertex with $i \leq j$, $i \leq (n-1) - j$ and $i, j \in \{t + 2, t + 3, \ldots, 3t\}$.

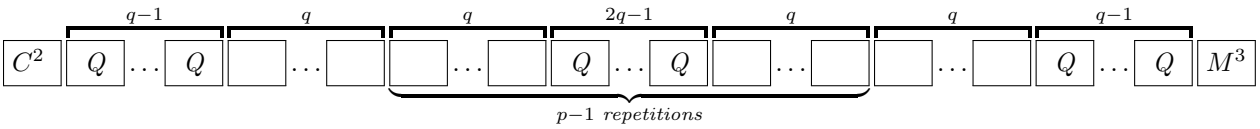
1. If $j - i < t + 1$, then the forward-diagonal neighbour $v_{0,j-i}$ of $v_{i,j}$ is in S.

2. If $j - i \geq t + 1$, then $j + i = 2i + (j - i) \geq 3(t + 1)3t + 3$, so the back-diagonal neighbour $v_{0,j+i}$ is in S.

Therefore, S is a dominating set. □

The pattern discovered by Sinko and Slater for the construction in Theorems 5.12 and 5.13 is a special case of a general pattern, which is given in Theorems 5.20 and 5.21.

Theorem 5.20. *Let* $p \geq 0$ *and* $q \geq 1$. *For all* $n = 2p(4q - 1) + 1$, *the rotated pattern*

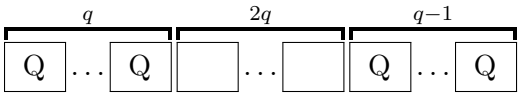


is an RBC of size $4p(2q - 1) + 1$.

Proof. The pattern is non-degenerate and uses two corner cells and three midpoint cells. Lemmas 5.4 and 5.6 imply that the remaining corner vertices and midpoint vertex would be redundant, which implies that we can assume, for the purpose of the proof, that all four corner vertices and all four midpoint vertices are present. Therefore, we can assume that the set S of vertices of Queen (n) corresponding to the rotated pattern above is perfectly symmetric about all rotations (including the corner and midpoint cells). As a result, it is sufficient to prove that every vertex on the forward and back diagonal of the board is

dominated, and that every vertex under both diagonals (all vertices $v_{i,j}$ in the triangle where $i \leq j$ and $i \leq (n-1)-j$) is dominated. The vertices on the diagonals are dominated by the corner cells in S.

Let $v_{i,j}$ be a cell with $i \leq j$ and $i \leq (n-1)-j$. Observe that the first $n-1$ columns of row 0 of the pattern consist of repeated instances of the following sequence of $4q-1$ cells



The pattern places a queen in every row and column with an index congruent to $-(q-1), -(q-2), \ldots, -1, 0, 1, \ldots, 1, \ldots, q-1,$ or q modulo $4q-1$. Therefore, if either i or j is congruent to any value in $\{-(q-1), \ldots, q\}$ modulo $4q-1$ then $v_{i,j}$ is dominated by a row or column neighbour.

Suppose $v_{i,j}$ does not fall into the case above. That is,

$$i \equiv k \pmod{4q-1} \text{ and } j \equiv \ell \pmod{4q-1}$$

where $k, \ell \in \{q, q+1, \ldots, (4q-1)-(q+1), (4q-1)-q = 3q-1\}$. Since $v_{i,j}$ lies in the triangle below the diagonal and back diagonal of the board, it has both a forward-diagonal neighbour and back-diagonal neighbour in row 0. Specifically, $v_{i,j}$ is a neighbour of $v_{0,j-i}$ and $v_{0,j+i}$. The two cases below establish that either one of these two vertices is in S, or the horizontal reflection of one of these vertices is in S (which, by Corollary 5.9, is sufficient to prove that $v_{i,j}$ is dominated).

1. If $\ell - k \in \{-(q-1), \ldots, q-1\}$, then $j-i \equiv -(q-1), \ldots, q-2,$ or $q-1 \pmod{4q-1}$, so the forward-diagonal neighbour $v_{0,j-i}$ is either in S or the reflection of a vertex in S.

2. Otherwise, $|\ell - k| \geq q$ (note that this inequality is not taken modulo $4q-1$). Let

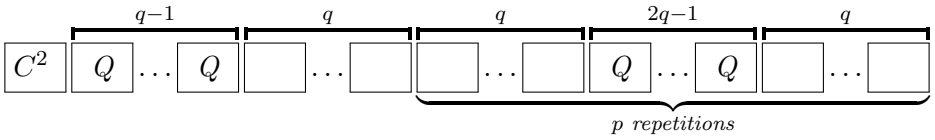
$a = \min(k, \ell)$ and $b = \max(k, \ell)$, and note that $|\ell - k| = b - a$ and $a + b = \ell + k$. Combining the bounds on k and ℓ given above with the fact that $b - a \leq q$ gives $q \leq a \leq 3q - 1 - (b - a) \leq 2q - 1$ and $2q \leq q + (b - a) \leq b \leq 3q - 1$. Therefore,

$$(4q - 1) - (q - 1) = 3q \leq a + b \leq 5q - 2 = (4q - 1) + (q - 1)$$

giving $a + b \equiv j + i \equiv -(q - 1), -(q - 2), \ldots, 0, \ldots, q - 2, q - 1 \pmod{4q - 1}$, so the back-diagonal neighbour $v_{0,j+i}$ is either in S or the reflection of a vertex in S.

□

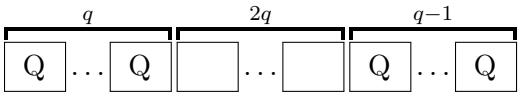
Theorem 5.21. *Let $p \geq 0$ and $q \geq 1$. For all $n = 2p(4q - 1) + 4q$, the rotated pattern*



is an RBC of size $(4p + 2)(2q - 1)$.

Proof. The pattern is non-degenerate and uses two corner cells. As in the proof of Theorem 5.20, we can assume that all four corner vertices are present, and therefore that the set S of vertices of Queen (n) corresponding to the rotated pattern above is perfectly symmetric about all rotations (including the corner cells). Similarly, all vertices on the main forward and back diagonals of the board are dominated by the corner cells, and therefore, by symmetry, it is sufficient to prove that every vertex in the triangle $v_{i,j}$ in the triangle where $i < j$ and $i < (n - 1) - j$ is dominated

Like Theorem 5.20, the first $n - 1$ columns of row 0 consist of repetitions of the $4q - 1$ cells



The proof therefore proceeds equivalently to that of Theorem 5.20. □

Theorems 5.15 - 5.19, together with the case where $p = 1$ in Theorem 5.20, give a general upper bound for the size of an RBC for all $n \pmod 8$. Theorem 5.22 combines these mutually exclusive cases.

Theorem 5.22. *Let $n \geq 8$ with $n \equiv k \pmod 8$ where $0 \leq k \leq 7$. Then the value $MinRBC(n)$ exists and is bounded above by the value in the table below (according to the value of k).*

k	Upper Bound	Justification
0	$n-2$	Theorem 5.18
1	$n-3$	Theorem 5.15
2	n	Theorem 5.19
3	$n-2$	Theorem 5.16
4	$n-2$	Theorem 5.18
5	$n-2$	Theorem 5.17
6	n	Theorem 5.19
7	$n-2$	Theorem 5.20 ($p = 1$)

□

5.5.1 Searching for Minimum RBCs

Because of the symmetry constraints on rotated vertices, the number of RBCs of Queen (n) is substantially less than the overall number of border dominating sets, so a computational search for minimum RBCs is more likely to succeed in a reasonable amount of time than one for minimum general border sets.

Since Framework 3.1 supports 'restricted domination' by allowing the initial candidate set C to be limited to any set of vertices, it is relatively easy to compute general border dominating sets using algorithms based on the Framework. However, modifying the initial

candidate set is not sufficient to generate RBCs. For example, an RBC can be characterized by the set of corner vertices, the set of midpoint vertices and the set of rotated vertices in row 0. Restricting the candidate set C for Framework 3.1 to these vertices will not produce an RBC, since when a rotated vertex in row 0 is added to a dominating set, all of its rotations must be added with it.

One way around this problem is to modify the algorithm to add all four rotations whenever a rotated vertex is used. However, implementing this approach requires a redesign of the bounding conditions, since all of the algorithms based on Framework 3.1 are designed to judge the viability of each potential dominating vertex based on the neighbourhood of that vertex: if adding a particular vertex also mandates adding three other vertices, the bounding conditions may no longer work.

To generate minimum RBCs, an alternative method was used which allowed standard restricted domination solvers, like those implementing Framework 3.1, to be used. Theorem 5.23 details the equivalence. The notation $N_H[v]$ is used to refer to the neighbourhood of a vertex v with respect to a particular graph H.

Theorem 5.23. *Let $n \geq 4$ and, for clarity, define $Q = Queen(n)$.*

Define a graph G with $V(G) = V(Q)$ and

$$E(G) = E(Q) \cup R_1 \cup R_2 \cup \ldots \cup R_{\lfloor n/2 \rfloor - 1}$$

where

$$R_j = \{v_{0,j}u : u \in N_Q[v_{j,n-1}] \cup N_Q[v_{n-1,(n-1)-j}] \cup N_Q[v_{(n-1)-j,0}]\} - \{v_{0,j}v_{0,j}\}.$$

Let $S = \{v_{0,j} : 0 < j < n/2\}$. If n is even, let $T = \{v_{0,0}, v_{0,n-1}\}$. If n is odd, let $T = \{v_{0,0}, v_{0,n-1}\} \cup \{v_{0,(n-1)/2}, v_{(n-1)/2,0}, v_{n-1,(n-1)/2}\}$. Finally, let $C = S \cup T$.

Then any dominating set D of G with $D \subseteq C$ corresponds to an RBC of size

$$|D \cap T| + 4|D \cap S|$$

of Q, and any RBC of Q corresponds to a dominating set $D \subseteq C$ of G.

Proof. Consider a dominating set $D \subseteq C$ of G, and let

$$\begin{aligned} D' = (D \cap T) &\cup (D \cap S) \\ &\cup \{v_{j,n-1} : v_{0,j} \in D \cap S\} \\ &\cup \{v_{n-1,(n-1)-j} : v_{0,j} \in D \cap S\} \\ &\cup \{v_{(n-1)-j,0} : v_{0,j} \in D \cap S\}. \end{aligned}$$

The vertices in $(D \cap T)$ correspond to corner and midpoint vertices of Q, and by the construction of T, the number of corner or midpoint vertices present cannot exceed the limits of Definition 5.10. All four of the rotations of each vertex $v_{0,j} \in D \cap S$ are added to D'. Therefore, the set D' meets the criteria for an RBC if D' dominates every vertex of Q.

Let $v_{i,j}$ be a vertex of Q such that $v \notin D'$ and let $u \in D$ be a dominator of $v_{i,j}$ in G (that is, a vertex such that $v_{i,j} \in N_G[u]$). If $u \in S$, then $v_{i,j} \in N_Q[u]$ since, by the definition of G, the neighbourhood of non-rotated vertices is the same as Q. If $u \in T$, then since $N_G[u]$ is defined to be the union of $N_Q[u']$ for each rotation u' of u, either $v_{i,j}$ is dominated by u itself in Q or $v_{i,j}$ is dominated by some rotation of u in Q. Since all rotations of u are present in D', $v_{i,j}$ is dominated by D' and therefore D' is a dominating set of Q.

Consider an RBC D' of Q and let

$$D = (D' \cap T) \cup (D' \cap \{v_{0,j} : 1 < j < n/2\}).$$

To demonstrate the correspondence in the theorem, D will be shown to be a dominating set

of G. Let $v_{i,j} \in V(Q) - D$, and let $u \in D'$ be a dominator of $v_{i,j}$ in Q. If $u \in T$, then $u \in D$, so $v_{i,j}$ is dominated by u in G. Otherwise, u is a rotated vertex. Let u' be the rotation of u that lies in row 0. By the definition of D, u' is in D and, since $v_{i,j} \notin D$, $v_{i,j} \neq u'$. By the definition of $E(G)$, $N_Q[u] \subseteq N_G[u']$, so $v_{i,j} \in N_G[u']$, and therefore $v_{i,j}$ is dominated by D in G. □

Dominating sets of the auxiliary graph G in Theorem 5.23 correspond to RBCs of Queen (n), but there is no assurance that a minimum dominating set of G corresponds to a minimum RBC. Therefore, to find the minimum size of an RBC, all dominating sets of the auxiliary graph G were generated (with the candidate set C restricted as given in the theorem) and, after applying the correspondence in the theorem, the minimum RBC was selected.

5.5.2 Summary of Border Queen Results

Table 5.5 summarizes all known results on the border domination problem for $1 \leq n \leq 100$. For $n \leq 29$, the value of bor (Queen (n)) has been found computationally (all values of bor (Queen (n)) for $n \geq 14$ are new results from this research). Table 5.5 does not contain results for bor (Queen (n)) for $n \geq 29$ since the exact value of bor (Queen (n)) is not yet known for those values of n. The table also includes the upper bounds on the size of an RBC produced by Theorems 5.20, 5.21 and 5.22. In cases where the size of a minimum RBC, or one of the upper bounds, matches the border domination number, the corresponding table entry is in red. In cases where the value of an upper bound matches the minimum size of an RBC, the corresponding table entry is in bold.

Table 5.5: Summary of Border Domination Parameters

n	bor	Min. RBC	Upper Bounds Thm. 5.20	Thm. 5.21	Thm. 5.22
1	1	**1**			-2
2	1	**1**			2
3	2	**2**			1
4	3	**3**			2
5	3	**3**			**3**
6	4	**4**			6
7	5	**5**	**5**		**5**
8	6	**6**			**6**
9	6	**6**			**6**
10	6	**6**		**6**	10
11	9	**9**			**9**
12	10	**10**			**10**
13	9	**9**	**9**		11
14	12	**12**			14
15	13	**13**	**13**		**13**
16	10	**10**		**10**	14
17	14	**14**			**14**
18	16	18			**18**
19	13	**13**	**13**		17
20	18	**18**			**18**
21	19	**19**			**19**
22	14	**14**		**14**	22
23	21	**21**	**21**		**21**
24	22	**22**			**22**
25	17	**17**	**17**		22
26	24	26			**26**
27	25	**25**			**25**
28	18	**18**		**18**	26
29	25	**25**	**25**		27
30		30			**30**
31		21	**21**		29
32		30			**30**
33		30			**30**
34		22		**22**	34
35		33			**33**
36		30		**30**	34
37		25	**25**		35
38		38			**38**
39		37	**37**		**37**
40		26		**26**	38
41		38			**38**
42		42			**42**
43		29	**29**		41
44		42			**42**
45		41	**41**		43
46		30		**30**	46
47		45	**45**		**45**
48		46			**46**
49		33	**33**		46
50		42		**42**	50

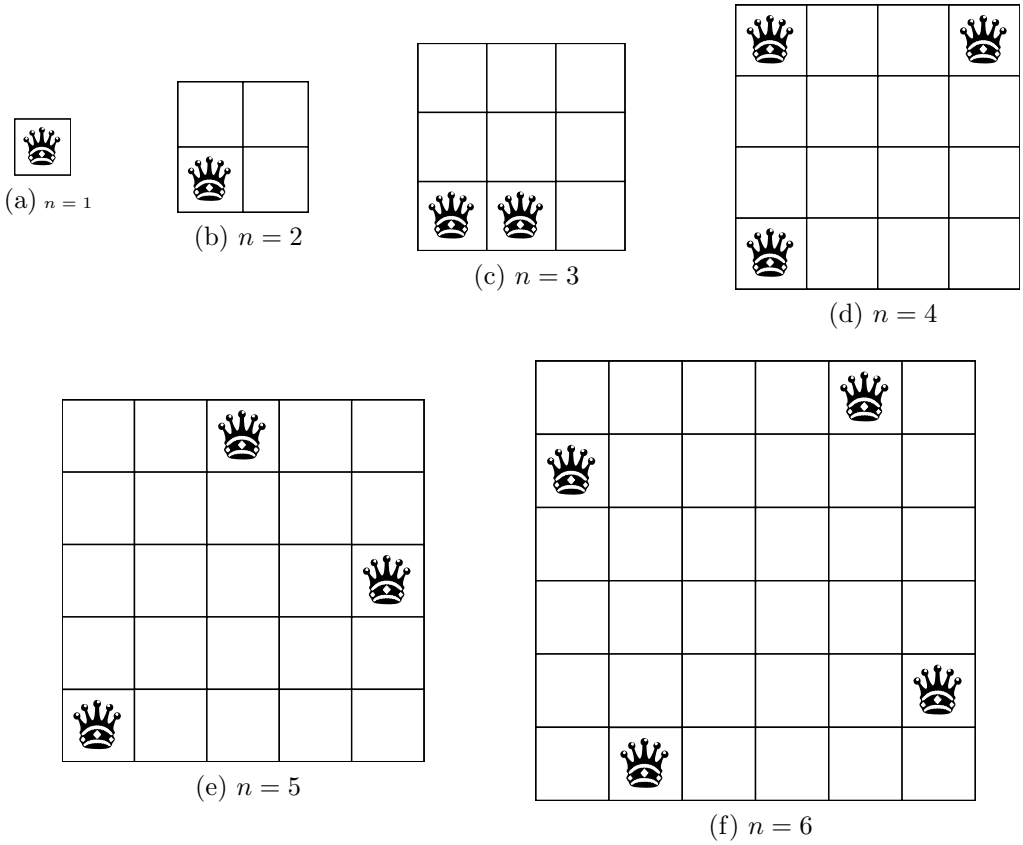
n	Min. RBC	Upper Bounds Thm. 5.20	Thm. 5.21	Thm. 5.22
51	49			**49**
52	34		**34**	50
53	51			**51**
54	54			**54**
55	37	**37**		53
56	50		**50**	54
57	49	**49**		54
58	38		**38**	58
59	57			**57**
60	58			**58**
61	41	**41**		59
62	62			**62**
63	61	**61**		**61**
64	42		**42**	62
65	62			**62**
66	66			**66**
67	45	**45**		65
68	66			**66**
69	67			**67**
70	46		**46**	70
71	61	**61**		69
72	70			**70**
73	49	**49**		70
74	74			**74**
75	73			**73**
76	50		**50**	74
77	73	**73**		75
78	66		**66**	78
79	53	**53**		77
80	78			**78**
81	78			**78**
82	54		**54**	82
83	81			**81**
84	82			**82**
85	57	**57**		83
86	86			**86**
87	85	**85**		**85**
88	58		**58**	86
89	81	**81**		86
90	90			**90**
91	61	**61**		89
92	78		**78**	90
93	89	**89**		91
94	62		**62**	94
95	93	**93**		**93**
96	90		**90**	94
97	65	**65**		94
98	98			**98**
99	85	**85**		97
100	66		**66**	98

The data in Table 5.5 shows that MinRBC(n) is equal to bor$($Queen$(n))$ for all but two of the known cases. Additionally, for all of the computed cases in the table except $n = 1, 2, 3, 4, 5, 6, 14$, MinRBC(n) is exactly equal to the best upper bound computed by Theorems 5.20, 5.21 and 5.22. In particular, in every case where Theorems 5.20 or 5.21 apply, the size of the RBC produced by the construction in those theorems for the minimum value of q is equal to the size of a minimum RBC and, for values of n where bor$($Queen$(n))$ is known, equal to bor$($Queen$(n))$. In general, the data in Table 5.5 seems to support the claim that a characterization of RBCs may lead to a characterization of bor$($Queen$(n))$, and additionally that the three theorems given earlier in this chapter come close to achieving such a characterization of MinRBC(n). The two aspects of the data which may contradict such a claim are the cases where MinRBC(n) does not match bor$($Queen$(n))$, and the cases where MinRBC(n) is not equal to one of the three bounds.

For $n = 1, 2, 3, 4, 5, 6, 14$, the size of a minimum RBC does not match the size of an RBC produced by any of Theorems 5.20, 5.21 and 5.22. For $n \leq 6$, this is due to the theorems not being applicable. For some of these values, all minimum RBCs are degenerate (and therefore, could not match any of the canonical RBCs produced by the bounding theorems). Figure 5.8 gives minimum RBCs for $n = 1$ through 6, and Figure 5.9 gives a minimum RBC for $n = 14$.

Of the minimum RBCs in Theorem 5.8, only the case $n = 6$ is non-degenerate and therefore admits a canonical form. Some of the other cases, in addition to being degenerate, also violate other aspects of Definition 5.10. For example, the minimum RBC for $n = 4$ contains three corner vertices.

For $n \geq 8$ (the minimum value of n for which all three bounding theorems can produce a bound), the only value of n for which one of the three bounds does not match MinRBC(n) is $n = 14$. The bound produced by Theorem 5.22, via Theorem 5.19, is 14, while the minimum RBC has size 12. In general, since the bound produced by Theorem 5.19 is n,



(a) $n = 1$ (b) $n = 2$ (c) $n = 3$ (d) $n = 4$ (e) $n = 5$ (f) $n = 6$

Figure 5.8: Examples of minimum RBCs for $n \in \{1, 2, 3, 4, 5, 6\}$.

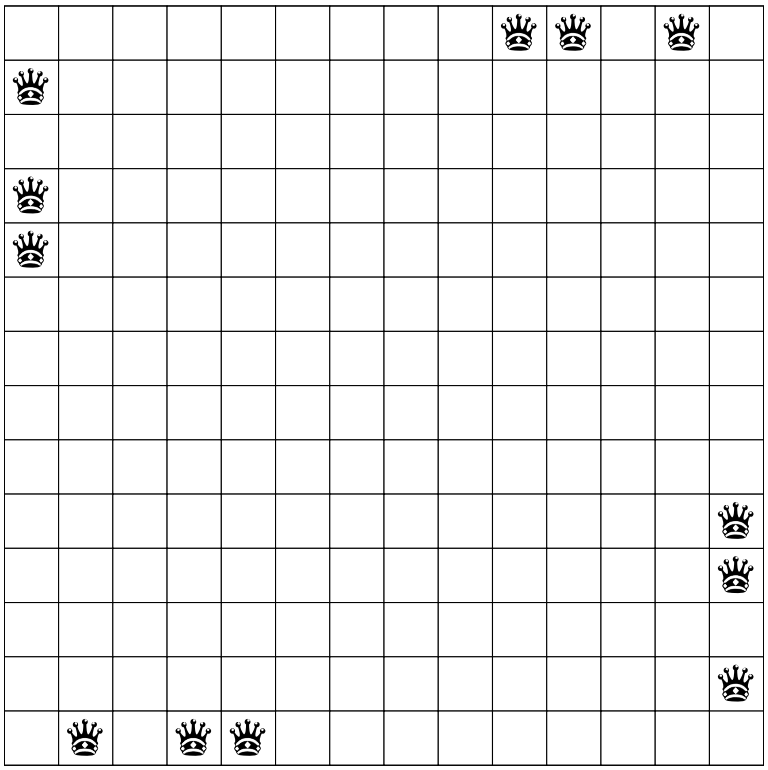


Figure 5.9: A minimum RBC of size 12 of Queen (14).

it will never match $\text{bor}(\text{Queen}(n))$ since Theorem 2.8 gives a construction for a non-RBC border dominating set of size $n-2$ for all $n \geq 4$. The two cases in Table 5.5 for which $\text{MinRBC}(n)$ does not match $\text{bor}(\text{Queen}(n))$ occur at $n = 18$ and $n = 26$, both of which correspond to cases where Theorem 5.19 produces the best upper bound. However, in those cases, $\text{MinRBC}(n) = n$.

The remaining cases where $\text{MinRBC}(n) = n$ occur at $n = 30, 38, 42, 54, 62, 66, 74, 86, 90$. All are cases where no bounds apply except Theorem 5.19, and it is clear that $\text{MinRBC}(n) \neq \text{bor}(\text{Queen}(n))$ for all such cases. Since Theorem 5.19 applies for $n = 4t+2$ for $t \geq 1$, all of the values in Table 5.5 for which $\text{MinRBC}(n)$ have $n \equiv 2 \pmod 4$, but many other values which are congruent to 2 modulo 4 are covered by Theorem 5.21. In general, Theorem 5.21 applies when $n = 2p(4q-1)+4q$ for some $p \geq 0$ and $q \geq 1$. This expression for n can be rewritten as $n = (2p+1)(4q-1)+1$, so $n-1$ is divisible by $4q-1$, which is congruent to 3 (mod 4). Theorem 5.21 applies in all cases where $4q-1$ divides n for some q. Since $n-1 \equiv 1 \pmod 4$, all of its divisors must be congruent to either 1 or 3 (mod 4), and Conjecture 5.24 proposes a characterization of the cases where Theorem 5.21 cannot apply and proposes that $\text{MinRBC}(n) = n$ (due to Theorem 5.19) in those cases.

Conjecture 5.24. *Let $n \geq 15$. If every odd prime divisor p of $n-1$ has $p \equiv 1 \pmod 4$, then $\text{MinRBC}(n) = n$.*

Note that if all prime divisors p of $n-1$ have $p \equiv 1 \pmod 4$, then since the set

$$S = \{k : k \equiv 1 \pmod 4\}$$

is closed under multiplication, every divisor of $n-1$ is also congruent to 1 (mod 4).

Conjecture 5.24 sets a lower bound of $n \geq 15$, since the conjecture would otherwise apply to the values $n = 6$ and $n = 14$. Although the $n = 6$ case can be relegated by assuming that (as with many of the theorems in this chapter) small values of n exhibit peculiar behavior,

the case $n = 14$ remains an outlier, both as a case where the conjecture does not apply and as the single case where $\text{MinRBC}(n)$ is not determined by one of the bounding theorems. In the data for $n \geq 15$, there is no evidence that the $n = 14$ case is an example of a recurring pattern, although such a pattern may appear at larger values of n than the ones covered by Table 5.5.

We conclude this section with a set of additional conjectures based on the data in Table 5.5. Conjecture 5.25 is the most cautious of the set, and proposes that Theorems 5.20, 5.21 can produce a minimum RBC for every value n that meet the initial conditions of those theorems. Except for the cases covered by Conjecture 5.24, $\text{MinRBC}(n)$ matches $\text{bor}(\text{Queen}(n))$ for all values of n for which $\text{bor}(\text{Queen}(n))$ is known and $\text{MinRBC}(n)$ is equal to the minimum of the three bounds in Theorems 5.20, 5.21 and 5.22. Conjectures 5.26, 5.27, 5.28 present possible resolutions to the open questions of the minimum size of RBCs and the minimum size of border dominating sets. Conjecture 5.27 is a weak form of Conjecture 5.28.

Conjecture 5.25. *Let $n \geq 1$.*

1. *If there exists a minimum q such that $n = 2p(4q-1)+1$ for some p, then the construction in Theorem 5.20 yields a minimum RBC and $MinRBC(n) = 4p(2q-1)+1$.*

2. *If there exists a minimum q such that $n = n = 2p(4q-1)+4q$ for some p, then the construction in Theorem 5.21 yields a minimum RBC and $MinRBC(n) = (4p+2)(2q-1)$.*

Note that the two cases above are mutually exclusive.

Conjecture 5.26. *For all $n \geq 15$, an RBC with size $MinRBC(n)$ can be constructed using one of Theorems 5.20, 5.21 and 5.22.*

Conjecture 5.27. *For all $n \geq 1$, $bor(Queen(n)) \geq MinRBC(n) - 2$.*

Conjecture 5.28. *Let $n \geq 1$. If $n \geq 15$ and every prime divisor p of $n-1$ has $p \equiv 1 \pmod 4$, then $bor(Queen(n)) = n-2$. Otherwise, $bor(Queen(n)) = MinRBC(n)$.*

Chapter 6

Unidom

The culmination of the research in Chapters 3 and 4 was the creation of a unified domination solver for arbitrary graphs, called `unidom`. The `unidom` program is designed to be fast, self-contained (with no reliance on outside libraries, besides the C++ standard library) and modular. The `unidom` program contains implementations of all of the dominating set algorithms described in this thesis. It also supports various input formats (and procedural graph generators) and preprocessing stages, and allows either optimization (finding a minimum dominating set) or exhaustive generation of dominating sets of a particular size.

This chapter contains a high level summary of `unidom`'s architecture and features. It is not intended to serve as the documentation for the software, only as a presentation of a research artifact.

6.1 The `unidom` Architecture

The `unidom` program is designed to serve as a research tool for solving instances of domination problems. For some applications, the 'stock' features of `unidom`, such as the algorithms described in this thesis, should suffice to find a solution. In other cases, it may be necessary to extend `unidom`'s functionality with new code to adapt it to a particular problem. To

accommodate both possibilities, `unidom` has been designed to be modular and allow easy extension at the source code level, but also to have a flexible interface to allow different features to be selected at the command line instead of recompiling or modifying the code. At a high level, the program operates on a 'pipeline' model, where an instance of a domination problem is passed through several stages, which may include manipulations like renumbering of vertices, graph transformations or the application of reduction rules. The relative isolation of each stage in the `unidom` pipeline also allows for easy modification by future contributors.

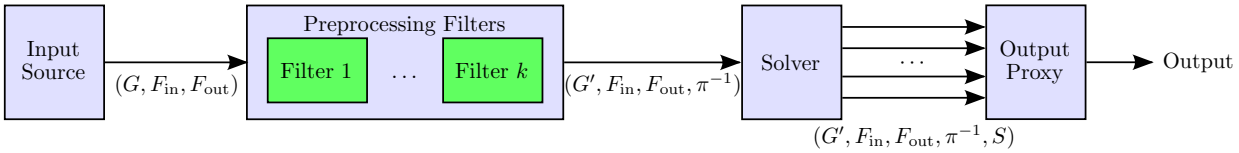


Figure 6.1: Diagram of the 'pipeline' used by `unidom` computations.

Figure 6.1 illustrates the high-level architecture of `unidom`. Sections 6.2 - 6.5 describe each stage of the pipeline. It is not always necessary to use all of the stages of the pipeline. For example, to use `unidom` to find a minimum dominating set of a graph stored in an input file '`graph.txt`' using the 'MDD_minCD' solver, the command line

```
$ ./unidom -S MDD_minCD < graph.txt
```

would suffice. On the other hand, command line options are available to configure each aspect of the pipeline, as shown in the diagram below, which gives an illustrated view of the command line to create a pipeline for generating a queen graph of order 8, renumbering the vertices by a BFS traversal, finding minimum dominating sets with the '`fixed_order`' solver, using an initial upper bound of 7, and outputting the best dominating set found.

```
$ ./unidom  -I queen -n 8  -F renumber_bfs  -S fixed_order -u 7  -O output_best
            Input Source   Filter           Solver              Output Proxy
```

The '`-I`', '`-F`', '`-S`' and '`-O`' options are used to specify the input source (described in Section 6.2), filter(s) (described in Section 6.3), solver (described in Section 6.4) and output proxy (described in Section 6.5), respectively. Multiple filters can be specified with multiple '`-F`' options. The first argument after a '`-I`', '`-F`', '`-S`' or '`-O`' flag must be the name of the component (input source/filter/solver/output proxy) to add to the pipeline. After the component name, component-specific arguments can be specified (as with the '`-n 8`' and '`-u 7`' in the example above). By convention, argument flags for components are lowercase. The arguments for a particular component must immediately follow that component's '`-I`', '`-F`', '`-S`' or '`-O`' flag.

6.2 Input Source

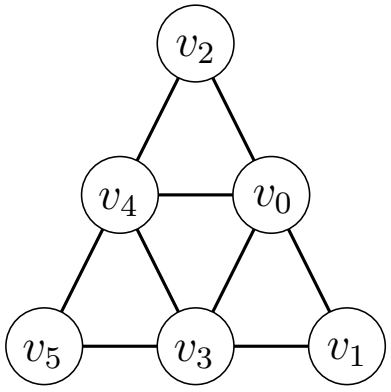
An input source in `unidom` produces a triple $(G, F_{\text{in}}, F_{\text{out}})$, where G is a graph (represented internally with an adjacency list) and $F_{\text{in}}, F_{\text{out}} \subseteq V(G)$ are sets of vertices which restrict the eventual dominating sets generated for G. All vertices in F_{in} must be added to any dominating set produced by the solver and any vertices in F_{out} must be excluded.

The default input source reads graphs from standard input in the text-based adjacency list format detailed below.

```
<number of vertices>
<degree of vertex 0> <neighbour 0 of vertex 0> <neighbour 1 of vertex 0> ...
<degree of vertex 1> <neighbour 0 of vertex 1> <neighbour 1 of vertex 1> ...
...
<degree of vertex n-1> <neighbour 0 of vertex n-1> <neighbour 1 of vertex n-1> ...
```

Figure 6.2 shows a graph (specifically, TG (3)) with numbered vertices and its representation in the format given above.

The text-based input format should be sufficient for most uses, but the interface for input sources allows procedural generators to be used as well. In cases where a parameterized



(a) Diagram

```
6
4 1 2 3 4
2 0 3
2 0 4
4 1 0 4 5
4 0 2 3 5
2 3 4
```

(b) Adjacency list

Figure 6.2: Example of the adjacency list representation of TG (3).

family of graphs is being studied, it may be easier to add a generator directly to `unidom` as an input source than to write a separate generator and feed the results into `unidom`. The sets F_{in} and F_{out} allow restricted domination problems (such as the border queen problem) to be encoded as `unidom` inputs.

The input sources implemented for this research are detailed below, with relevant parameters. An input source is selected with the `-I` parameter.

`basic_input`	Read adjacency lists from standard input.
`border_queen`	Generate Queen (n) (where n is specified by the `-n` parameter), with all interior vertices added to F_{out}.
`code_graph`	Generate the covering code graph $\text{Code}_r(n, q)$, with the parameters n and r specified via `-n` and `-r` (respectively) and the parameter q specified by `-base`.
`hexrook`	Generate HR (n) (where n is specified by the `-n` parameter).
`kneser`	Generate Kneser (n, k), with n and k specified by the `-n` and `-k` parameters, respectively.
`queen`	Generate Queen (n) (where n is specified by the `-n` parameter).
`TG`	Generate the triangular grid graph TG (n) (where n is specified by the `-n` parameter).

6.3 Preprocessing Filters

After the input graph is generated, any number of preprocessing filters can be used to apply various transformations to the $(G, F_{\text{in}}, F_{\text{out}})$ triple before the solver is invoked. Filters may modify the input triple in any way, although filters for the general domination problem should normally be constrained to operations such that the dominating sets which are eventually output will be in one-to-one correspondence with those of the original input graph G. As Chapters 3 and 4 showed, renumbering of vertices is often useful during a dominating set computation, and several filters have been added to facilitate this. To ensure that the generated dominating sets can be mapped back onto the original input graph, the output of the preprocessing stage is a tuple $(G', F_{\text{in}}, F_{\text{out}}, \pi^{-1})$, where π^{-1} is a permutation of $\{1, 2, \ldots, |V(G)|\}$ such that, for a vertex $v_i \in V(G')$ $\pi^{-1}(i)$ gives the index of vertex v_i in the original graph G (prior to the preprocessing stage). The permutation π^{-1} is then carried through each remaining stage of the pipeline so that it can be used to generate output

consistent with the original input graph.

Filters are specified with the `-F` parameter, and are applied in the same order they appear on the command line. The standard set of available filters is detailed below.

`force_in` Given a list of vertex indices on the command line after the '`-F force_in`' parameter (for example '`-F force_in 0 2 6 10`'), add the specified vertices to the set F_{in}. This will result in the produced dominating sets being required to contain all of the specified vertices.

`force_out` Given a list of vertex indices on the command line after the '`-F force_out`' parameter (for example '`-F force_out 0 2 6 10`'), add the specified vertices to the set F_{out}. This will result in the produced dominating sets being required to exclude all of the specified vertices (or, if no such sets exist, no dominating sets being generated at all).

`renumber_bfs` Renumber the vertices of the graph to correspond to the order in which vertices are visited by a breadth-first search traversal starting at vertex 0. A different choice of root can be specified by the `-root` parameter.

`renumber_maxdeg` Renumber the vertices of the graph in ascending order by degree (so vertex 0 will have maximum degree and vertex $n-1$ will have minimum degree).

`renumber_mindeg` Renumber the vertices of the graph in ascending order by degree (so vertex 0 will have minimum degree and vertex $n-1$ will have maximum degree).

`renumber_random` Renumber the n vertices of the graph by applying a random permutation of $\{0, 1, \ldots, n-1\}$. The permutation is generated with a Knuth shuffle algorithm [46], which selects permutations uniformly at random under the assumption that the underlying pseudorandom generator produces uniformly distributed random numbers. The underlying pseudorandom generator is the Mersenne Twister implementation provided by the C++ standard library.

6.4 Solver

The solver is specified with the `-S` parameter and takes the $(G', F_{\text{in}}, F_{\text{out}}, \pi^{-1})$ tuple from the preprocessing stage as its input. Solvers generate one or more dominating sets $S \subseteq V(G')$ and provide each of them to the output stage as part of a tuple $(G', F_{\text{in}}, F_{\text{out}}, \pi^{-1}, S)$. There is no requirement that solvers generate only minimum (or even minimal) dominating sets. Some of the provided solvers are designed to exhaustively generate dominating sets with certain properties and others are designed to output only those sets which may be of minimum size (usually by outputting a set only when it is smaller than any previously generated set). Solvers are generally required to respect the sets F_{in} and F_{out} which may restrict the set of possible solutions. If a solver is incompatible with restricted domination, it is expected to generate an error message and terminate the program if F_{in} or F_{out} is non-empty, rather than simply ignoring the two sets and proceeding.

For problems in which dominating sets with very particular properties are needed (such as RBCs for the border queen problem), it is often easier (if potentially slower) to use a general exhaustive solver coupled with a customized output proxy which ignores all solutions which do not meet the criteria. A list of all solvers is detailed below; since it is not practical (due to code size limitations) to include all of the variants covered by the experimental data in Chapter 4, only a handful of high quality solvers have been included. If no solver is specified, the `fixed_order` variant is used. All of the solvers based on Framework 3.1 have a common set of configuration options and are provided in two versions: one to solve the optimization problem of finding a minimum dominating set and one to exhaustively generate dominating sets. The exhaustive generation variant of each solver has the suffix '`_all`'. For exhaustive generation, it may be useful to constrain the minimum and maximum size of the generated sets; the various configuration options for the solvers are described later in this section.

`DD` `DD_all`	The 'default' variant of Algorithm 3.5. Based on the experimental data in Chapter 4, this is an alias of the `DD_minCD_asc` solver.
`DD_minCD_asc` `DD_minCD_asc_all`	An implementation of Algorithm 3.5 using minimum CD for vertex selection, ascending neighbour ordering and with neither of the force stop or bound rechecking optimizations enabled.
`DD_minCD_desc` `DD_minCD_desc_all`	An implementation of Algorithm 3.5 using minimum CD for vertex selection, descending neighbour ordering and with neither of the force stop or bound rechecking optimizations enabled.
`MDD` `MDD_all`	The 'default' variant of Algorithm 3.7. Based on the experimental data in Chapter 4, this is an alias of the `MDD_minCD` solver.
`MDD_minCD_asc` `MDD_minCD_asc_all`	An implementation of Algorithm 3.7 using minimum CD for vertex selection, ascending neighbour ordering and with the force stop optimization disabled but bound rechecking enabled.
`MDD_minCD_desc` `MDD_minCD_desc_all`	An implementation of Algorithm 3.7 using minimum CD for vertex selection, descending neighbour ordering and with the force stop optimization disabled but bound rechecking enabled.
`MDD_minMDD_desc` `MDD_minMDD_desc_all`	An implementation of Algorithm 3.7 using minimum MDD for vertex selection, descending neighbour ordering and with the force stop optimization disabled but bound rechecking enabled.
`fixed_order` `fixed_order_all`	An implementation of Algorithm 3.2. This solver should be used on very small graphs, due to its low overhead. It should normally be combined with a renumbering filter (such as `renumber_bfs`) to improve performance (as demonstrated by the experiments in Chapter 4).

All of the solvers above share a common set of configuration options (which should be specified on the command line after the '`-S <solver>`' parameters), which are summarized

below.

`-u <upper bound>`	Restrict the computation to dominating sets of size at most `<upper bound>`. This is equivalent to setting the `desired_size` parameter in Framework 3.1.
`-l <lower bound>`	Restrict the computation to dominating sets of size at least `<lower bound>`. For optimizing solvers, this setting will cause the solver to immediately terminate after finding a set of size `<lower bound>`. For solvers which exhaustively generate dominating sets, this setting (possibly combined with the `-u` option) can be used to limit the range of sizes of the generated sets.

`-res`
`-mod`
`-resmod_depth`
	Set the residue, modulus and split depth for the res/mod approach for multiprocessor computation (see Section 5.2 for information).

6.5 Output Proxy

The output phase of the pipeline is interleaved with the solver's computation: whenever the solver produces a dominating set, the output phase is invoked. Output is managed by an 'output proxy' which determines how (and if) the dominating set should be output. The output phase can also contain verification (to ensure that the dominating set is valid, or that the restricted domination constraints have been obeyed). Formally, the output proxy receives a tuple $(G', F_{\text{in}}, F_{\text{out}}, \pi^{-1}, S)$ from the solver, where S is a dominating set and the remaining items are as defined in previous sections. In the `unidom` implementations solvers based on Framework 3.1, the output phase is invoked every time a dominating set

smaller than the current best is found (line 4 of Framework 3.1) in optimization mode, or, in exhaustive generation mode, every time a dominating set has been found (line 6 of Framework 3.1). The output proxy may output every certificate it receives, wait until the end of the computation and output the best overall certificate, or output some subset of certificates which meet certain conditions. This allows unusual variants of domination problems to be easily implemented in `unidom` (at the expense of running time) by using an exhaustive generation solver to generate all dominating sets of a particular size and defining a custom output proxy to filter the sets according to the problem requirements.

The set S in the $(G', F_{\text{in}}, F_{\text{out}}, \pi^{-1}, S)$-tuple provided to the output proxy is a subset of the vertices of G', which may not correspond directly to the original input graph. To output a certificate in terms of the original input graph, the permutation π^{-1} can be used to map the contents of S back to the vertices of the original input graph G. All of the provided output proxies perform this mapping.

The provided output proxies are detailed below. If no output proxy is explicitly specified, the '`output_all`' output proxy is used.

`output_all` Print every dominating set received from the solver to standard output (after applying the renumbering permutation π^{-1}), followed by '-1' printed on a line by itself after the computation ends. Each dominating set is printed on its own line and consists of the size of the set followed by a listing of the vertices in the set.

`output_best` At the end of the computation, print the minimum-size dominating set received over the course of the computation. If multiple dominating sets were received with the minimum size, the first one is output. Sets are printed in the same format as the '`output_all`' solver, including the '-1' printed after the computation ends. If the '`-graph`' parameter is used, a representation of the graph (in the adjacency list format described in Section 6.2) is printed before the dominating set (this feature may be useful for generating certificates for solved cases of the domination problem on particular graphs). If the '`-gamma`' parameter is used, only the size of the minimum dominating set is printed, not the set itself.

`queen_board` Only usable in conjunction with the '`queen`' or '`border_queen`' input sources. At the end of the computation, outputs the minimum dominating set of a queen graph with a chessboard-style representation, with 'Q' indicating the placement of queens and '_' indicating empty cells. The '`-all`' parameter can be used to enable output of all sets received, instead of just the minimum size set.

Chapter 7

Conclusions and Future Research

The main contributions made by this thesis were the backtracking framework and derived algorithms (Chapter 3), the newly solved open cases of the queen domination, independent domination and border domination problems (summarized in the introduction to Chapter 5), the definition and theoretical results for rotated border constructions (Section 5.5) and the development of the `unidom` research tool for solving domination problems (Chapter 6). As observed in Sections 1.2 and 1.4, there is a paucity of research regarding general purpose dominating set solvers, other than wrappers around solvers for other hard problems (such as integer programming). The solvers created for this thesis have already been successfully used to solve previously open problems, but, as the data in Chapter 4 shows, the newly developed algorithms are not universally better than existing solvers. The author hopes that the research in this thesis can be used as a foundation for the development of higher quality solvers in the future, and that the experimental data collected for this thesis can be of use to researchers evaluating prospective algorithmic approaches. The `unidom` program is intended to be the first step toward a dedicated suite of domination solvers, in the tradition of SAT solvers or integer programming solvers.

One of the immediate avenues for future algorithmic research is the use of symmetry

in the domination computation. Even within this thesis, many of the studied graphs had large symmetry groups, and a practical method to eliminate isomorphic solutions from the computation would likely result in a significant speedup. Simple heuristic methods to reduce isomorphism are often easy to implement (for example, by constraining the first few vertices selected for a dominating set based on known properties of particular graphs), but may be graph- or family-dependent. General isomorphism reduction requires knowledge of the automorphism group of the graph, as well as a practical method for using the group elements such that the overhead incurred does not outweigh the speedup of isomorphism removal. Practical algorithms to compute automorphism groups (as a set of generators) are already widely used (notably, the `nauty` program [49]). Previous research by Myrvold and Fowler [50] and Bird and Myrvold [5] has presented practical approaches to isomorphism elimination in backtracking searches.

The experiments of Chapter 4 showed that the SageMath solver had significantly better performance than any of the new algorithms on certain classes of graphs, namely the triangle grid graphs and the cartesian products of cycles. Further study is needed to identify the cause of the performance difference on these classes. Given that the triangle grid graphs are subgraphs of the hex rook graphs (on which the new algorithms handily outperformed SageMath), a question for future research would be whether the difference is due to the total number of edges (or some related metric, such as average degree). One possible cause is that, in triangle grid graphs, the closed neighbourhoods of some vertices are completely contained in the closed neighbourhoods of other vertices. In an integer/linear programming solver, such cases create linear dependencies between the constraints of the program, which may allow the solver to eliminate some constraints.

The smallest open case of the queen domination and independent domination problems is now $n = 26$. This case should be within the reach of the algorithm used for the open cases in this thesis with currently available research computing clusters (which for funding

reasons were not leveraged for this research). Such clusters typically have thousands of processors, and all of the infrastructure for massive parallelization is already available in the implementations presented by this research (including the `unidom` program). One avenue that was not explored in this research was the behavior of the domination number of non-square queen graphs (for example, a 10×20 chessboard), which might lead to a classification of generalized queen graphs. In addition to queen graphs, several other families of graphs might benefit from the new algorithms, including (rectangular) grid graphs (for which the domination number is known, but the structure of minimum dominating sets is still studied) and Cartesian products of graphs (as part of the search for a counterexample to Vizing's conjecture).

As far as the queen problem is concerned, the overall goal of using computers to solve open cases is to provide more data for theoretical research with an eye toward developing conjectures and, ultimately, finding a theoretical solution. This symbiosis between computation and theory is more evident in the border queen results presented in Chapter 5.5. Previous research was conducted using observations from a small number of known values of $\mathrm{bor}\left(\mathrm{Queen}\left(n\right)\right)$. Using the newly computed minimum border dominating sets for $n \leq 29$, the theory of rotated border constructions was developed, and Theorems 5.20, 5.21 and 5.22 were proven. The theoretical results on RBCs were used to compute the minimum size of an RBC for $n \leq 100$ using the transformation described in Section 5.5.1. These computational results, in turn, provided evidence that the constructions in Theorems 5.20, 5.21 and 5.22 yield minimum RBCs for all $n \leq 100$, and additionally, that the minimum size of an RBC matches the border domination number in almost all known cases. The culmination of this progression is the set of conjectures stated in Section 5.5.2 on the potential resolutions to the border domination problem. Besides bringing together the theoretical material in Section 5.5, the conjectures at the end of Chapter 5 also tie together all of the other facets of this thesis, since without the algorithms and experiments of previous chapters, the data which

led to the conjectures would not have been available. For future research, the most salient open question posed by this thesis is Conjecture 5.28: For values of $n \geq 15$ where at least one odd prime divisor of $n-1$ is not congruent to 1 $\pmod 4$, is it the case that $\mathrm{MinRBC}(n)$ is always equal to $\mathrm{bor}(\mathrm{Queen}(n))$?

Bibliography

[1] SageMath website (http://www.sagemath.org). Retrieved June 13, 2016.

[2] Jochen Alber, Hongbing Fan, Michael R. Fellows, Henning Fernau, Rolf Niedermeier, Fran Rosamond, and Ulrike Stege. *Refined Search Tree Technique for Dominating Set on Planar Graphs*, pages 111–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[3] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, Cambridge, 2009.

[4] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1 – 31, 2009.

[5] William Bird and Wendy Myrvold. Generation of colourings and distinguishing colourings of graphs. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures*, volume 9214 of *Lecture Notes in Computer Science*, pages 79–90. Springer International Publishing, 2015.

[6] Béla Bollobás and Ernest J. Cockayne. The irredundance number and maximum degree of a graph. *Discrete Mathematics*, 49(2):197–199, 1984.

[7] Boštjan Brešar, Paul Dorbec, Wayne Goddard, Bert L. Hartnell, Michael A. Henning, Sandi Klavžar, and Douglas F. Rall. Vizing's conjecture: a survey and recent results. *Journal of Graph Theory*, 69(1):46–76, 2012.

[8] David Brink. The inverse football pool problem. *Journal of Integer Sequences*, 14(8):Article 11.8.8, 9, 2011.

[9] P. A. Burchett. On the border queens problem and k-tuple domination on the rooks graph. *Congressus Numerantium*, 209:179–187, 2011.

[10] A. P. Burger, E. J. Cockayne, and C. M. Mynhardt. Domination and irredundance in the Queens' graph. *Discrete Mathematics*, 163(13):47 – 66, 1997.

[11] A. P. Burger and C. M. Mynhardt. An improved upper bound for Queens domination numbers. *Discrete Mathematics*, 266(1-3):119–131, 2003. The 18th British Combinatorial Conference (Brighton, 2001).

[12] Alewyn Petrus Burger. *The queen's domination problem*. PhD thesis, University of South Africa, 1998.

[13] M. Chlebík and J. Chlebková. Approximation hardness of dominating set problems in bounded degree graphs. *Information and Computation*, 206(11):1264 – 1275, 2008.

[14] E. J. Cockayne. Chessboard domination problems. *Discrete Mathematics*, 86(1):13 – 20, 1990.

[15] E. J. Cockayne, O. Favaron, C. Payan, and A. G. Thomason. Contributions to the theory of domination, independence and irredundance in graphs. *Discrete Mathematics*, 33(3):249 – 258, 1981.

[16] E. J. Cockayne, S. T. Hedetniemi, and D. J. Miller. Properties of hereditary hypergraphs and middle graphs. *Canadian Mathematical Bulletin*, 21(4):461–468, 1978.

[17] E. J. Cockayne and C. M. Mynhardt. Properties of Queens graphs and the irredundance number of Q_7. *Australasian Journal of Combinatorics*, 23:285–300, 2001.

[18] Gérard Cohen, Iiro Honkala, Simon Litsyn, and Antoine Lobstein. *Covering codes*, volume 54 of *North-Holland Mathematical Library*. North-Holland Publishing Co., Amsterdam, 1997.

[19] Peter Damaschke. Irredundance number versus domination number. *Discrete Mathematics*, 89(1):101 – 104, 1991.

[20] Joe DeMaio and Hong Lien Tran. Domination and independence on a triangular honeycomb chessboard. *The College Mathematics Journal*, 44(4):307–314, 2013.

[21] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, London, 2013.

[22] Rodney G. Downey, Michael R. Fellows, Catherine McCartin, and Frances Rosamond. Parameterized approximation of dominating set problems. *Information Processing Letters*, 109(1):68 – 70, 2008.

[23] J. Ellis, H. Fan, and M. Fellows. The dominating set problem is fixed parameter tractable for graphs of bounded genus. *Journal of Algorithms*, 52(2):152 – 168, 2004.

[24] Odile Favaron, Gerd H. Fricke, Dan Pritikin, and Jol Puech. Irredundance and domination in Kings graphs. *Discrete Mathematics*, 262(1):131 – 147, 2003.

[25] Henning Fernau. Minimum dominating set of queens: A trivial programming exercise? *Discrete Applied Mathematics*, 158(4):308 – 318, 2010. 6th Cologne/Twente Workshop on Graphs and Combinatorial Optimization (CTW 2007).

[26] Dmitry Finozhenok and William D. Weakley. An improved lower bound for domination numbers of the Queen's graph. *The Australasian Journal of Combinatorics*, 37:295–300, 2007.

[27] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Measure and conquer: domination–A case study. In *International Colloquium on Automata, Languages, and Programming*, pages 191–203. Springer, 2005.

[28] Fedor V. Fomin and Dimitrios M. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speed-up. *SIAM Journal on Computing*, 36(2):281–309, 2006.

[29] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[30] Chris Godsil and Gordon Royle. *Algebraic graph theory*, volume 207 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2001.

[31] Igor Gorodezky. Dominating sets in Kneser graphs. Master's thesis, University of Waterloo, 2007.

[32] Fabrizio Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209 – 214, 2006.

[33] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

[34] Magnús M. Halldórsson. Approximating the minimum maximal independence number. *Information Processing Letters*, 46(4):169 – 172, 1993.

[35] S. T. Hedetniemi and R. C. Laskar. Bibliography on domination in graphs and some basic definitions of domination parameters. In S.T. Hedetniemi and R. C. Laskar, editors, *Topics on Domination*, volume 48 of *Annals of Discrete Mathematics*, pages 257 – 277. Elsevier, 1991.

[36] Juraj Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer-Verlag, Berlin/Heidelberg, 2013.

[37] Robert W. Irving. On approximating the minimum independent dominating set. *Information Processing Letters*, 37(4):197 – 200, 1991.

[38] Yoichi Iwata. A faster algorithm for dominating set analyzed by the potential method. In *International Symposium on Parameterized and Exact Computation*, pages 41–54. Springer, 2011.

[39] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[40] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[41] Petteri Kaski and Patric R. J. Östergård. *Classification algorithms for codes and designs*, volume 15 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2006.

[42] Matthew D. Kearse and Peter B. Gibbons. Computational methods and new results for chessboard problems. *The Australasian Journal of Combinatorics*, 23:253–284, 2001.

[43] Matthew D. Kearse and Peter B. Gibbons. A new lower bound on upper irredundance in the Queens' graph. *Discrete Mathematics*, 256(12):225 – 242, 2002.

[44] Gerzson Kéri. Tables for bounds on covering codes. `http://www.sztaki.hu/~keri/codes/`. Accessed on May 21, 2017.

[45] Sandi Klavžar and Norbert Seifter. Dominating cartesian products of cycles. *Discrete Applied Mathematics*, 59(2):129 – 136, 1995.

[46] Donald E. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley, Reading, Massachusetts, 2011.

[47] Emil Kolev and Tsonka Baicheva. About the inverse football pool problem for 9 games. In *Seventh International Workshop on Optimal Codes and Related Topics*, pages 125–133, 2013.

[48] Jeff Linderoth, François Margot, and Greg Thain. Improving bounds on the football pool problem by integer programming and high-throughput computing. *INFORMS Journal on Computing*, 21(3):445–457, 2009.

[49] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[50] Wendy Myrvold and Patrick Fowler. Fast enumeration of all independent sets of a graph up to isomorphism. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 85:173–194, May 2013.

[51] Patric R. J. Östergård, Zehui Shao, and Xiaodong Xu. Bounds on the domination number of Kneser graphs. *Ars Mathematica Contemporanea*, 9(2):197–205, 2015.

[52] Patric R. J. Östergård and William D. Weakley. Values of domination numbers of the queen's graph. *Electronic Journal of Combinatorics*, 8(R29):1–19, 2001.

[53] Geevarghese Philip, Venkatesh Raman, and Somnath Sikdar. Solving dominating set in larger classes of graphs: FPT algorithms and polynomial kernels. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 694–705. Springer Berlin Heidelberg, 2009.

[54] Dieter Rautenbach. On the differences between the upper irredundance, upper domination and independence numbers of a graph. *Discrete Mathematics*, 203(13):239 – 252, 1999.

[55] W. W. Rouse Ball. *Mathematical Recreations and Essays (10th ed.)*. London,Macmillan, 1922.

[56] Anne Sinko and Peter J. Slater. Queen's domination using border squares and (a, b)-restricted domination. *Discrete Mathematics*, 308(20):4822 – 4828, 2008.

[57] N. J. A. Sloane. Online encyclopedia of ingeger sequences (OEIS): Entry A094087 (domination number of cartesian product of n-cycles). Retrieved June 15, 2016.

[58] N. J. A. Sloane. Online encyclopedia of ingeger sequences (OEIS): Entry A229803 (domination number of hex rook graph). Retrieved June 15, 2016.

[59] N. J. A. Sloane. Online encyclopedia of ingeger sequences (OEIS): Entry A251419 (domination number of triangle grid graph). Retrieved June 15, 2016.

[60] Johan M. M. van Rooij and Hans L. Bodlaender. Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147 – 2164, 2011.

[61] Lutz Volkmann and Vadim E. Zverovich. Proof of a conjecture on irredundance perfect graphs. *Journal of Graph Theory*, 41(4):292–306, 2002.

[62] Stan Wagon. Graph theory problems from hexagonal and traditional chess. *The College Mathematics Journal*, 45(04):278–287, 2014.

[63] Peng-Jun Wan, K. M. Alzoubi, and O. Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1597–1604, 2002.

[64] William D. Weakley. Upper bounds for domination numbers of the Queen's graph. *Discrete Mathematics*, 242(13):229 – 243, 2002.

[65] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall Inc., Upper Saddle River, New Jersey, 1996.

[66] Jie Wu, Ming Gao, and I. Stojmenovic. On calculating power-aware connected dominating sets for efficient routing in ad hoc wireless networks. In *International Conference on Parallel Processing, 2001*, pages 346–354, 2001.

[67] Vadim E. Zverovich. On the differences of the independence, domination and irredundance parameters of a graph. *Australasian Journal of Combinatorics*, 27:175–186, 2003.