Complexity of Counting N-Queens

Irfansha Shaik

Submitted to Swansea University in fulfillment of the requirements for
the Degree of MSc by Research, Computer Science

Swansea University
2019

Abstract

The N-Queens problem is the 150-year-old classical mathematical problem of placing N queens on an $N \times N$ chessboard such that no two queens share the same row, column or a diagonal, i.e. non-attacking queens. The counting N-Queens is the task of counting all such solutions of the N-Queens problem.

Despite being a widely researched problem and having simple constraints, the existence of efficient counting algorithms is not evident. Initial attempts to solve this problem using backtracking algorithmic approaches, i.e. enumerating all the solutions by backtracking, do not scale beyond a small N. Even with further improvements, i.e. using symmetry breaking and hardware acceleration techniques, the algorithms are still not scalable. So the critical question remains, does an efficient algorithm exist for the counting N-Queens problem?

To answer this question, we investigate the underlying backtracking trees abstracting away all the implementation details. We introduce and investigate the different classes of Queens Branching trees (QBT trees) that are generated by the efficient propositional satisifiability (SAT) based implementations with propagation. In particular, we consider a subclass of QBT trees called AmoAlo trees with additional unsatisfiability test and propagation. We conjecture that these trees have polynomial tree sizes in the size of output. To investigate the conjectures, we developed `GenericSAT`, a SAT-based implementation with different branching heuristics. Among these branching heuristics, we show that the *Taw* heuristics have smaller trees suggesting the existence of algorithms with output-polynomial runtime. We discuss the state-of-the-art counting approaches and compare the trees generated to our branching trees. We also discuss the possibility of understanding unsatisfiability in the N-Queens problem using visualisation. Further, we discuss the possibility of going beyond enumeration paradigm for the counting N-Queens problem from the observations made and propose an idea of using `Cube-and-Conquer` approach.

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)
Date ....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)
Date ....

Statement 2

I hereby give my consent for my thesis, if accepted to be made available online in the Universitys Open Access Repository and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)
Date ....

Contents

Acknowledgements

First, to my supervisor, Dr. Oliver Kullmann, thank you for all the hours you put in the meetings and the guidance throughout the project. To my other supervisor, Dr. Arno Pauly, and my friends Ankit, Connor, Michael, Safal, Shashank, Swetha, and especially Arved, thank you for reviewing this thesis and providing valuable suggestions. To all the staff, especially Sara, thank you for your prompt responses and efficient handling of the formal procedures.

To my friend and mentor, Hoda, thank you for helping me adjust in the starting of my degree and for all your support. To my friends and colleagues, Aled, Ali, Alharbi, Avishek, Dylan, Elif, Emilio, Jade, Jay, Manduhu, Omniah, Stavros and Thomas (in alphabetical order) to name a few, thank you for all the little chats and lunches we had together, you all had a positive impact on my research. To my flatmates, AJ, George, Klaus, James, Tom, Will and especially Jack, thank you for being supportive and helping me concentrate on the project. To my dear friend Kashinka, thank you so much for being there when I needed the most. To my parents, thank you for the financial support without which this degree would not have been possible.

Finally, I want to dedicate this work in the loving memory of my maths teacher, Venkateswara Rao. I would not be here without the guidance and support from mentors like you; you were a good mentor and a kind person.

List of Figures and Tables

List of Abbreviations

AI	Artificial Intelligence
ALO	At-Least-One
AMO	At-Most-One
AmoAlo	At-Most-One and At-Least-One
BCP	Boolean Clause Propagation
CC	Component Caching
CDCL	Conflict Driven Clause Learning
CDP	Counting by Davis-Putnam
CNF	Conjunctive Normal Form
CPU	central processing unit
CSP	constraint satisfaction problem
#CSP	Counting Constraint Satisfaction Problem
d-DNNF	Deterministic Decomposable Negation Normal Form
DLL	Davis, Logemann, and Loveland
DNF	Disjunctive Normal Form
DNNF	Decomposable Negation Normal Form
domoverwdeg	domover weight based degree
DPLL	Davis, Putnam, Logemann, Loveland
DPP	Davis-Putnam Procedure
FPGAs	field programmable gate arrays
GAC	Generalised Arc Consistency
HC	Hybrid Coding scheme
HCOP	Hybrid Coding Omitting binary clauses Packing scheme

IBCP	Implicit Boolean Clause Propagation
ldf	largest domain first
MP	Mathematical Programming
NNF	Negation Normal Form
NP	Non-Deterministic polynomial time
OBDDs	Ordered Binary Decision Diagrams
PAC	Probabilistic Arc Consistency
PH	Polynomial Hierarchy
PP	Pre-Processing
QBT	Queens Branching Trees
RAM	Random Access Machine
SAT	Propositional Satisfiability
#SAT	Counting Propositional Satisfiability
sdf	smallest domain first
UP	Unit Propagation
VSIDS	Variable State Independent Decaying Sum
wdeg	weight based degree

Chapter 1

Introduction

The 8-Queens puzzle is the classical mathematical problem of placing 8 Queens on an 8×8 chessboard such that no two queens share the same row, column, or a diagonal. Figure 1.1 is an example solution for the 8-Queens problem.

Max Bezzel first proposed this problem in 1848 in the Berliner Schachzeitung [5], and later Lionnet generalised it to an arbitrary N in 1876 in the Nouvelles Annales de Mathematiques [41]. Counting all the solutions for an $N \times N$ board is the counting N-Queens problem. Despite having simple constraints, counting all the solutions is a hard combinatorial problem with exponential solution-space, and it belongs to the counting class #P and its completeness is unclear when starting with an empty board.

Beginning from Gauss (see Paper [11] for the history of the 8-Queens problem) attempting to count 8-Queens solutions (by hand) to the recent 27-Queens computation by Preußer [47] using `field programmable gate arrays` (FPGAs), the main focus is to understand the solution space and provide efficient algorithms for counting all the solutions. Initially, the main research problem was to understand the solution space and possibly provide a closed form mathematical function that gives solutions count directly, the point of interest slowly shifted to just counting (enumerating) all the solutions using efficient algorithms as the solution space is too *complex* to understand as of now. While the N-Queens problem in itself does not have direct applications, being a combinatorial problem which is often solved by backtracking approaches, any advances here would have implications in the problems that are solved using backtracking. Also counting problems, in general, have a wide range of applications, we discuss some of them briefly in Chapter 2. As we model the N-Queens problem suitable for propositional satisfiability (SAT) based implementations, the advances in our implementations can be applied to related SAT implementations and vice versa.

1.1 Challenges

A natural way to count all the solutions of the N-Queens problem is by going through all possible combinations. However, being a combinatorial problem, the state space increases exponentially, and it is easy to see that this simple

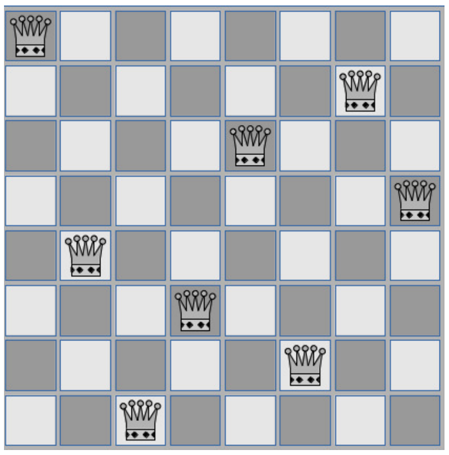

Figure 1.1: A solution to the 8-Queens puzzle

approach is infeasible. On the other hand, backtracking algorithms perform much better as we can cut-off exploring unnecessary state space. However, from the literature, it is evident that even the best-optimised backtracking algorithms do not scale well for $N \geq 24$. We discuss one such efficient implementation, `Somers'` algorithm in Subsection 5.1.1. Further advancements are made in the state-of-the-art for counting N-Queens solutions by reducing the search space using symmetry breaking and improving the speed of computation by hardware acceleration. Even with these improvements only up to $N = 27$ could be counted (see Paper [47] for the computation discussion) as of now due to the inherent challenges of backtracking approaches. The thousands of CPU hours spent counting solutions for these larger problems raise an essential question: *Do efficient algorithms exist for counting* N-Queens*?* Throughout the literature, our past implementations only look at the *speed* of computations or the run-time of the algorithm. While run-time is an essential factor but because the solutions (and state space) grow exponentially, we currently cannot look beyond small N. A natural way to answer these questions is to study the solution space theoretically. However, to our knowledge, no proper theory exists for a theoretical investigation despite being a widely researched problem. In this work, we attempt to address these challenges.

1.2 Possible solutions

The specific implementations and hardware details of the algorithms that solve counting N-Queens problem usually make it hard to investigate their complexity. For a theoretical investigation, we abstract away all the implementation and hardware details and consider the underlying backtracking trees. We argue that by studying the sizes of trees and their growth will provide sufficient insight into the complexity of the algorithms. Studying the complexity of counting problem directly, at least for the N-Queens problem, does not seem to be straight forward. Moreover, all the approaches (we encountered) seem to enumerate all the solutions instead of counting. So we first consider enumeration of the N-Queens problem and investigate its complexity (more discussion in Section 2.4).

We earlier mentioned that even efficient backtracking algorithms do not scale well beyond small N. So we consider SAT based algorithms, in particular, we look at branching backtracking algorithms which are a subclass of backtracking algorithms. While the worst case complexity of these algorithms is the same as any other subclass of backtracking algorithms, they tend to perform exponentially better on practical instances. The main improvements in these algorithms seem to be from intelligent branching heuristics and propagation. So we want to investigate if these implementations perform well on the counting N-Queens problem and perform better than the current state-of-the-art. We conjecture that simple `DLL` (a basic SAT algorithm, further discussed in Subsection 2.4.1) based algorithms are equally good as the other approaches. So we consider the underlying trees that are produced in the `DLL` like algorithms and investigate their complexity.

We mathematically model the N-Queens problem to allow SAT based implementations. Using this modelling, we introduce different classes of branching trees or splitting trees (similar to the trees produced by `DLL` algorithms) called Queens branching trees. The strengths of the SAT based implementations are unsatisfiability detection and propagation. So we introduce subclasses of these Queens branching trees with unsatisfiability and propagation as parameters. In particular, we consider a subclass of trees called AmoAlo trees with the unsatisfiability and propagation that naturally arise from the constraints of the problem.

While the statements on tree growth of some basic classes of trees are straight forward, they are challenging to implement (or not efficient for implementation). On the other hand, the subclasses of trees such as AmoAlo trees are relatively easy to implement and are efficient algorithms. However, we do not have any statements on the sizes of AmoAlo trees and their growth. To investigate this, we developed `GenericSAT`, a SAT based framework in C++ with branching heuristics specific for the N-Queens problem. In a class of trees, branching heuristics usually have a significant impact on the tree sizes. So we implement and discuss different branching heuristics specific to the N-Queens problem and their effect on tree sizes.

We also look at other approaches such as #SAT solvers, #CSP solvers, Knowledge compilers and specialised solvers to investigate their underlying trees. Comparing these different trees might provide an insight into the existence of efficient algorithms. Further, we investigate if some algorithms produce trees with sizes that are asymptotically smaller than the solutions count. Existence of such algorithms, in fact, represents the existence of counting algorithms instead of enumeration.

1.3 Overview

In Chapter 2, we formally define the counting problem and briefly discuss their applications. We also discuss the complexity of counting problems in general and then briefly look at two main branches exact and approximate counting.

Here we conjecture that counting for N-Queens is as hard as enumerating all the solutions and discuss `DLL` algorithms for the enumeration.

In Chapter 3, we mathematically model the N-Queens problem and introduce different classes of Queens branching trees for investigating the complexity of the counting problem. We model the problem such that SAT based implementations are possible. In particular, we consider a subclass of Queens branching trees, AmoAlo trees, i.e., the Queens branching trees with AmoAlo unsatisfiability and propagation for implementation. Further, we state our main research questions, such as the existence of algorithms with output-polynomial run-time and make some conjectures on them.

In Chapter 4, we investigate AmoAlo trees introduced in Chapter 3. Here we provide details of our `GenericSAT`, software a SAT based implementation developed to investigate our conjectures on complexity questions in particular for AmoAlo trees. A point to be noted is that the main focus of `GenericSAT` framework is to allow prototype development for investigating our complexity conjectures. However, we also attempt to provide an efficient implementation for counting all the solutions. The choice of branching heuristics significantly affects the tree sizes, and we investigate different branching heuristics for answering complexity questions in AmoAlo trees, as our primary focus is on tree sizes and their growth. We provide implementations for an existing heuristics called *Taw* heuristics and 3 other heuristics that we developed specifically to the N-Queens problem. Then we discuss the data produced by these different heuristics and provide empirical argumentation strengthening our conjectures.

In Chapter 5, we look at other state-of-the-art counting approaches and compare the tree sizes to estimate their relative complexity. In particular, we consider one efficient implementation from specialised algorithms, #CSP, #SAT and knowledge compilers for comparison. We argue that if we can generate smaller trees without too much overhead, our algorithms will eventually perform better with the increase in N. So we try to compare the growth of tree sizes or similar comparable properties in different trees. While we compare these different trees, the main point of investigation is to analyse if the tree sizes are polynomial in the size of output.

In Chapter 6, we discuss our ongoing work, i.e., our approach to understanding the unsatisfiability and propagation better in an attempt to prove the conjectures. Finally, we discuss the possibility of going beyond enumeration using `Cube-and-Conquer`, a hybrid approach, in future work.

Chapter 2

Counting

In an NP problem, the decision problem is: *does a solution exists for a specific input?* and counting problem is: *how many solutions exist for a given input?* We formally define the counting problem:

Definition 2.0.1 *Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation, its* ***counting problem*** *is:* for some input $x \in \{0,1\}^*$, how many $y \in \{0,1\}^*$ exist such that $(x, y) \in R$?

Counting has many applications in Artificial intelligence (AI), Statistical Physics and Guiding search heuristics. The Bayesian belief problems in AI can be formulated into counting problems. Dempster-Shafer theory (presented in [56]) is widely used to handle uncertain information in AI systems. Orponen [44] proves that the Dempster's rule of combination used in Dempster-Shafer theory can be reduced to a counting problem #SAT (see Definition 2.0.2). Roth [52] shows that considering the counting problem for knowledge representation with Bayesian belief networks instead of propositional satisfiability problem might be a better approach. Darwiche [14] shows that d-DNNF (further explanation on d-DNNF in Subsection 5.4.1) for the model counting can be used for efficiently predicting the behaviour of broken devices. Kumar [38] uses the model counting for characterisation of diagnoses. Burton [10] and Lebowitz [39] discuss some applications of counting in Statistical Physics. Another important application for counting is to use it as part of guiding search heuristics where finding a solution is very hard (NP-hard problems in general). Horsch [29] provides a Probabilistic Arc Consistency (PAC) algorithm to solve Constraint Satisfaction Problems (CSP) using the probabilities of occurrence of variables in the solutions. Here, the probabilities are computed by estimating the number of solutions. Similarly, counting is used to guide the search heuristics in constraint satisfaction problems (see Papers [34, 49, 19, 63] for further discussions) whereas Iwama [30] used counting for testing CNF satisfiability.

There are several approaches to solving counting problems. For example, specialised solvers are developed to solve specific problems. The `Somers'` algorithm discussed in Subsection 5.1.1 is one such algorithm for the N-Queens problem. The problems can also be solved by either counting through constraint satisfaction (#CSP) (discussed further in Section 5.3) or propositional satisfiability (#SAT) (discussed further in 5.2). The constraints of the problems are formulated in some (specific) representation and are solved using generic solvers. Alternatively, there are knowledge compilers (further discussed in 5.4) that compile the problem into a target language such that counting can be done easily. Most applications for counting problems do not need an exact count, but an approximate count is sufficient. Hence it is natural to consider approximate counting to solve slightly larger problems. We discuss exact and approximate counting in more detail in the Sections 2.2 and 2.3 respectively. Further discussions on model counting are available in the Handbook of Satisfiability [6].

In this thesis for counting N-Queens, we mainly focus on propositional model counting or #SAT. We do not use the existing solvers but develop a SAT based implementation for solving chessboard related problems. #SAT is counting distinct satisfying assignments of a given propositional formula. We formally define #SAT:

Definition 2.0.2 *Let f be a propositional boolean formula,* **#SAT** *is counting the total number of unique truth assignments that satisfy the formula f.*

Due to an exponential increase in state space, the #SAT solvers do not scale well beyond a few hundred variables compared to SAT solvers, which can handle millions of variables. The hardness of #SAT is further discussed in Section 2.1. The initial attempts of exact model counters are based on extending the `DPLL` algorithm (based on [18]) to counting. However, many advances in SAT solving cannot be directly used in counting as the techniques are specialised to narrow down to a single solution quickly. The `DPLL` based counters enumerate the solutions by exploring the complete solution space resulting in a lack of scalability. Some approaches, such as counting using connected components (discussed in Subsection 5.2.1) initially implemented in `relsat` [32] perform well in sparse problems by finding the clusters. However, the moderately constrained problems which tend to have solutions finely spread over the solution-space are relatively more difficult. In other words, the hardness of counting problems need not depend on the number of solutions. In our initial investigations, we observed the same in counting N-Queens and counting N-Rooks, a closely related chessboard problem. Despite having more solutions, the N-Rooks counting seems to be easier compared to the N-Queens counting in `DPLL` solvers.

2.1 Complexity

Solving a counting problem also solves its corresponding decision problem, i.e., by simply checking if the total solution count is non-zero. So counting problems are at least as hard as their corresponding decision problems and belong to #P-class introduced by Valiant [62].

Definition 2.1.1 *Let $f : \{0,1\}^* \rightarrow \mathbb{N}$ be a function and M be a polynomial-time algorithm, the function f is in* ***#P-class*** *if there is a polynomial transformation function $p : \mathbb{N} \rightarrow \mathbb{N}$ such that for every input $x \in \{0,1\}^*$:*

$$f(x) = |y \in \{0,1\}^{p(|x|)} : M(x,y) = 1|$$

In any class of problems, a problem is called **complete** if every other problem in that class can be reduced to it.

Definition 2.1.2 *A function f is* ***#P-complete****, if f is in #P and every problem in #P-class (see Definition 2.1.1) can be reduced to f in polynomial time.*

Using Cook-Levin reduction [12], #SAT can be shown to be #P-complete (see Paper [2] for complete proof). By #P-Completeness of #SAT, Toda [59] showed that it is at least as hard as solving a quantified boolean formula (QBF) with a fixed number of universal and existential quantifiers on its variables. The other counting problems are proved to be #P-complete by reducing them to #SAT. Here reduction means transforming one problem into another, often to use the existing algorithms. Often the reductions for counting problems are parsimonious, i.e., the number of solutions is preserved from one problem to another. In SAT solving, the problems are usually represented in conjunctive normal form (CNF) and disjunctive normal form (DNF).

Definition 2.1.3 *Let S be a set of literals, a propositional formulae f is in* ***conjunctive normal form*** *(*CNF*) over S if it is a conjunction of clauses and each clause is a disjunction of literals in S.*

Definition 2.1.4 *Let S be a set of literals, a propositional formulae f is in* ***disjunctive normal form*** *(*DNF*) over S if it is a disjunction of clauses and each clause is a conjunction of literals in S.*

The counting assignments in CNF (see Definition 2.1.3) and DNF (see Definition 2.1.4) is proved to be #P-complete. Similarly, many other related problems such as counting cycles in a graph are proved to be #P-complete (see Papers [48, 61, 62] for proofs and further discussions). From Toda's theorem (presented in [60]), given an oracle to a #P-complete problem, any problem in the polynomial hierarchy (PH) can be solved in polynomial time.

$$(PH) \subseteq P^{\#\text{SAT}}$$

It is intuitive that the problems with NP-complete decision problems belong to #P-complete and are hard to compute. Surprisingly Valiant [61] showed that some problems with polynomial-time decision problems such as 2-SAT, Horn-SAT can still be #P-complete and are equally hard to solve. Regarding the complexity of approximate counting, the estimates with high quality and confidence often depend on the near-uniform sampling of solution-space, which is much harder than generating a single solution. Roth [52] further discusses the hardness of approximate and exact counting for most restricted languages including `2MONOCNF`, `2HORN`, and `3u-2HORN`. Roth also proves that for some problems, even with an easy decision problem approximating is NP-hard.

2.2 Exact counting

Davis and Putnam [18] provided a procedure for computing quantification theory and proposed an algorithmic implementation popularly known as the `DPLL` algorithm. Initial exact model counters are based on Counting by Davis-Putnam (`CDP`) algorithm (see Paper [8] for the algorithm) which is based on the Davis-Putnam procedure (`DPP`) (presented in [18]). Birnbaum [8] shows that a formula F with m clauses on n variables has an average runtime of $\mathrm{O}(n \cdot m^d)$, where p is the probability of any literal in any given clause and $d = \lceil \frac{-1}{log_2(1-p)} \rceil$. This average runtime presents the possibility that despite the worst-case complexity of #SAT, the hard instances do not occur that often in practice. The basic concept of `CDP` algorithms is to branch on a chosen variable and apply unit propagation if available. Here the branching heuristics applied to choose a branching variable has a significant impact on tree sizes. There are some families of unsatisfiable problems where every `DPLL` tree results in exponential size when a fixed ordering is used whereas quasi-polynomial sized search trees exist for dynamic ordering (see Papers [9, 3] for the investigations). Branching heuristics for the counting N-Queens problem are discussed in Subsection 4.2.2.

Partial assignments are considered for avoiding unnecessary computation, i.e., if a propositional formula f is satisfied and has k unset variables then any assignment of the k variables will result in a satisfying solution. For counting, instead of expanding the subtree fully, which is quite impractical, we can count all the resulting solutions directly, which is equal to 2^k.

Two classes of algorithms, Look-ahead solvers and Conflict Driven Clause Learning (`CDCL`) solvers, arise from the basic `DPLL` approach. Look-ahead solvers are mainly based on choosing a branching variable by looking ahead, i.e., choosing the best possible variable by *looking at* each available variable. `CDCL` solvers learn from conflicts that arise in the search tree. Variations are implemented on backtracking such as chronological and non-chronological backtracking depending on the level to which they backtrack. These solvers perform well on the problems where local information is essential.

The Advances in SAT solvers that are suitable for counting are later applied to #SAT solvers. Bayardo and Schrag [33] presented the idea of using look-back techniques to guide the search. Later, counting through connected components (proposed in [32]) are implemented in `relsat` to efficiently count the solutions. Component caching is proposed in `cachet` (see Paper [53] for discussion on implementations) for reusing the already assigned parts of the formula. While it is useful, larger instances require large cache memory, reducing its effectiveness. Modern #SAT solvers use techniques such as component caching, implicit boolean constraint propagation (implicit BCP), preprocessing and other techniques to achieve efficiency. These advances are inherited into `sharpSAT` and improved upon to produce an efficient #SAT solver.

2.3 Approximate counting

Approximate counting is often used when exact counting is hard or exact solution count is not required. Most exact model counters such as `DPLL` based counters do not scale well due to the exploration of complete (exponentially large) search space. However, applications often do not need an exact count; for example, Horsch [29] shows that an approximate estimate of solution count is sufficient for guiding search heuristics. Jerrum [31] showed that the counting approximation and (efficient) uniform random generation of a problem are closely related. There are two directions of focus on approximation algorithms. One is providing fast counts without any bounds on the accuracy, and the other is to provide estimates with lower or upper bounds, but these are usually harder.

The fast approximate counts are generally provided by sampling from the set of solutions. The accuracy of exact count estimate depends on the uniformness of the sampling, and near-uniform sampling of the solution space is harder than finding a single solution. `ApproxCount` [64] uses Markov Chain Monte Carlo (MCMC) sampling for approximating solution count. While `ApproxCount` gives fast estimations and scales better than other exact model counters, it does not provide any guarantee on the estimate due to the lack of near-uniform sampling. `SampleCount` [26] provides lower bounds of the exact count with high confidence irrespective of the quality of the sampling process. It achieves provable lower bounds by choosing the order of variables such that the search tree is balanced. An exact counting solver such as `relsat` can also be used to obtain a lower bound by obtaining the exact solution count of smaller search space. An estimate on total count can also be obtained by scaling the solutions found in smaller search space to complete search space. `MBound` [27] uses a different approach for approximate counting that can use any SAT solver as a base where the accuracy increases with the efficiency of the SAT solver. Here, the strategy to achieve efficiency is to make the problem more constrained and to reduce the search space while maintaining the solution count. The efficiency is achieved by partitioning and constraining the search space such that more constraint propagation is achieved. This is related to *streamlining constraints* (see Paper [28] for further discussions), XOR-streamlining is used in `MBound` implementation.

In this dissertation, we focus on the exact counters directly, as the approximation (by regression) from existing solutions count already gives a reasonable estimate of the number of solutions and existing approximate solvers do not seem to scale beyond small N.

2.4 Counting and enumeration

Enumeration problem is listing all the solutions in some order. One way to count all the solutions for a problem is by enumerating all of them. The relative hardness of Counting class and Enumeration classes is not clear. Intuitively for any problem counting might seem easier than enumeration but there is no clear argumentation to prove this. In our initial investigations for the N-Queens problem, every solver seems to enumerate all the solutions instead of counting. Also, theoretically investigating the complexity of counting algorithms for the N-Queens problem is not clear. On the other hand, investigating (both theoretically and empirically) some subclasses of branching algorithms that enumerate all the solutions is relatively straight forward. So we first consider enumerating all the solutions for the N-Queens problem instead of counting. We conjecture that counting for the N-Queens problem is (asymptotically) as hard as enumerating all the solutions.

Conjecture 2.4.1 *Counting* N-Queens *is equally hard as enumerating all the solutions.*

Assuming the Conjecture 2.4.1 is true, we further conjecture that `DLL` algorithms (further discussed in Subsection 2.4.1), which are simple branching algorithms, are better than `DPLL` solvers (with advanced techniques) due to lack of significant improvement by additional clause learning, component caching and implicit BCP techniques (detailed explanation of these techniques is provided in 5.2.2).

Conjecture 2.4.2 *Simple `DLL` algorithms with the boolean constraint propagation are equally good as any `DPLL` algorithms for* N-Queens*.*

Our algorithmic implementations for counting all the N-Queens solutions are similar to classical `DLL` algorithms.

2.4.1 DLL algorithms

Davis, Logemann, and Loveland or simply `DLL` algorithm (presented in [17]) is the first practical complete algorithm for propositional satisfiability problem or SAT. Let f be a propositional formula over n variables. SAT is decidable, and a trivial way of proving is by checking all the 2^n possible combinations over the variables. Checking all possible combinations is a complete algorithm but quite impractical due to the exponential growth. The `DLL` algorithm, on the other hand, improves over the naive approach by branching over variables and eliminating unit-clauses. We briefly discuss the `DLL` algorithm implementation here.

Let F be a CNF over n variables and C be some clause in F. Here, `DLL` algorithm 1 is presented as a recursive algorithm. It accepts a boolean formula F and returns if the formula is satisfiable or unsatisfiable. In the Line 2, F is unsatisfiable if it has an empty clause. In the Line 3, F is satisfied if every clause is satisfied, i.e., F is empty. In the Lines 5 to 7, unit propagation is applied iteratively and satisfying the unit clauses by forcing their literals. In the Lines 9 to 11, a branching variable b is chosen and `DLL` function is iteratively called with $(F|b)$ and $(F|\bar{b})$. If found satisfiable in any of the two branches, satisfiable is returned else unsatisfiable is returned. Choosing the right branching variable reduces the search tree size significantly. For SAT problems, the order of choosing the sub-trees also plays an important role. The satisfying assignment can also be returned by storing the variable assignments. In the case of a

Algorithm 1 DLL for propositional satisfiability

```
    Input: Boolean formula $F$ in CNF
    Output: Returns if $F$ is satisfiable or not
 1: function DLL
 2:     if $C = \emptyset$ is a clause in $F$ then return $\bot$
 3:     if $F = \emptyset$ then return $\top$
 4: Unit-propagate:
 5:     while $C \in F$ and $|C| = 1$ do
 6:         if $x \in C$ is positive literal then set $x =$ true in $F$
 7:         else set $x =$ false in $F$
 8: Branching:
 9:     choose branching variable $b$
10:     if DLL($F|b$) is $\top$ then return $\top$
11:     if DLL($F|\bar{b}$) is $\top$ then return $\top$
12:     return $\bot$
```

partial satisfying assignment, any random assignment of unset variables along with partial assignment will give a complete satisfying assignment. Now the same algorithm can be used to count all the solutions by not stopping after a single solution is found. First, we discuss the basic counting implementation and then about using partial assignments for efficiency.

Algorithm 2 DLL for counting

```
    Input: Boolean formula $F$ in CNF
    Output: Returns total satisfying assignments count
 1: function DLL
 2:     if $C = \emptyset$ is a clause in F then return 0
 3:     if $F = \emptyset$ and all variables are assigned to some value then return 1
 4: Unit-propagate:
 5:     while $C \in F$ and $|C| = 1$ do
 6:         if $x \in C$ is positive literal then set $x =$ true in $F$
 7:         else set $x =$ false in $F$
 8: Branching:
 9:     choose branching variable $b$
10:     return DLL($F|b$) + DLL($F|\bar{b}$)
```

In the algorithm 2, unit-propagation and branching are similar to basic DLL algorithm 1. The only difference is that we count all the solutions returned by each sub-tree (see Line 10). Further, once F is satisfied by a partial assignment with k unset variables we need not assign the rest of the variables. Instead, we return 2^k in the Line 3. The improvement from partial assignments is implemented in Counting Davis-Putnam or CDP algorithm (presented in [8]) and is the base of modern #SAT solvers. As discussed earlier, choosing the right branching variable plays a vital role in the counting as well. However, the order of choosing sub-trees does not matter in counting all solutions, as both the sub-trees have to be explored.

In our implementations for N-Queens problem (see Chapter 4), we observe that every solution found is a complete assignment; thus we use basic `DLL` counting algorithm without partial assignments.

Chapter 3

The N-Queens problem

The N-Queens problem is placing N queens on an $N \times N$ chessboard such that no two queens share the same row, column or diagonals. For example a solution of 4-Queens problem is shown in the Figure 3.1. The counting N-Queens problem is counting all such solutions for an $N \times N$ chessboard. The completion problem of the N-Queens problem is proved to be NP-complete (see Paper [25] for proof) and naturally the counting problem is expected to be at least as hard as NP-complete problems. The counting version belongs to the counting class #P. Bell and Stevens [4] provide a comprehensive survey on known results for the N-Queens problem. As discussed in Chapter 1, a proper study of counting N-Queens complexity might provide an insight into finding efficient algorithms.

In this chapter, we mathematically model the N-Queens problem and introduce a class of trees to study the complexity of the counting problem. The trees we consider here are a subclass of backtracking trees known as *branching* trees or *splitting* trees. We abstract away all the implementation and hardware details from the algorithms and study the complexity in terms of tree sizes and their growth. Assuming there is no significant overhead, the algorithms that produce smaller trees scale better. Modern generalised solvers, for example, SAT solvers implement many advanced techniques which might not improve the computation but only cause additional overhead. As we later look at the trees produced by `sharpSAT` with some advanced options we see that even though the trees are relatively smaller, the overhead is significant making the computation slower (presented in Subsection 5.2.3). To investigate the complexity of algorithms in terms of sizes of trees and their growth, we introduce a class of trees called Queens branching trees (QBT trees) which are essentially branching trees (similar to the trees produced by `DLL` algorithms).

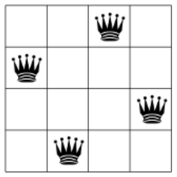

Figure 3.1: A solution to 4-Queens puzzle

In particular, we discuss a subclass of QBT trees called AmoAlo trees which have a certain level of unsatisfiability and propagation. Finally, we discuss the main research problems and provide some conjectures for different sub-classes of trees and attempt to answer their complexity in terms of tree sizes.

3.1 Preliminaries

In this section, we provide some basic definitions and notations for trees.

Definition 3.1.1 *A finite* ***rooted tree*** *is a pair* (T, r) *where*

1. T *is a directed graph (away from the root).*
2. $\mathrm{V}(T)$ *is the set of vertices and* $\mathrm{E}(T)$ *is the set of edges.*
3. *root* $\mathbf{rt}(\boldsymbol{T}) := r \in \mathrm{V}(T)$ *(the node with* 0 *indegree).*
4. *Let* $v, v' \in \mathrm{V}(T)$*.* v *is* ***parent*** *of* v' *and* v' *is a* ***child*** *of* v *if there is a directed edge from* v *to* v'*.*
5. *Every node* $w \in \mathrm{V}(T)\backslash\{\mathrm{rt}(T)\}$ *has a unique parent (the only in-neighbour).*
6. ***Leaves*** *are the vertices with no children (i.e., no out-neighbour) and are denoted by* $\mathrm{lvs}(T)$*.*
7. ***Inner-nodes*** *are the vertices other than leaves and are denoted by* $\mathrm{inds}(T)$*.*

In a rooted tree, we have $\mathrm{inds}(T) \cup \mathrm{lvs}(T) = \mathrm{V}(T)$. We use $\#\mathrm{lvs}(T) := |\mathrm{lvs}(T)|$ and $\#\mathrm{nds}(T) := |\mathrm{V}(T)|$ to denote the number of leaves and number of nodes respectively. The depth $\mathrm{d}_T(v)$ of a vertex $v \in \mathrm{V}(T)$ is the *distance* from $\mathrm{rt}(T)$.

Let $v \in \mathrm{V}(T)$ and (T_v, v) be some rooted tree. The tree T_v is a sub-tree of T, if $\mathrm{V}(T_v) \subset \mathrm{V}(T)$ and $\mathrm{E}(T_v) \subset \mathrm{E}(T)$. For example, 3.2 is a rooted tree with 1 as root. We only discuss rooted trees, so from here on any tree is a rooted tree unless otherwise specified.

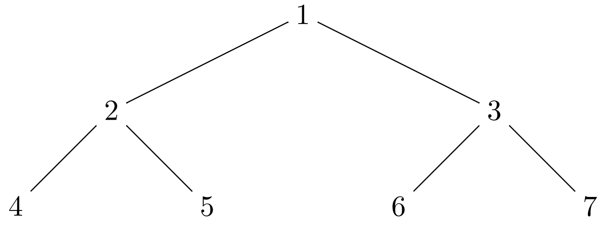

Figure 3.2: A rooted tree T with $\mathrm{rt}(T) = 1$, $\mathrm{inds}(T) = \{1, 2, 3\}$, and $\mathrm{lvs}(T) = \{4, 5, 6, 7\}$.

We now define a binary tree as recursive structure (provided in [21]), these are essentially rooted trees with at most two children.

Definition 3.1.2 *A **binary tree** is a 3-tuple* (L, S, R) *where*

1. *L, R are binary trees (or empty trees).*
2. *S is a node label usually an singleton set.*

Here the L and R are left and right subtrees respectively where S the label of the node.

Definition 3.1.3 *A **full binary tree** is a binary tree with no empty left and right subtrees for all inner nodes.*

For example, the tree 3.2 is also a full binary tree. The Queens branching trees that we introduce are full binary trees as we do not allow single child for the inner nodes. A point to be noted is that for the Queens branching trees, we label the nodes separately and consider the label S defined in the binary tree definition as an empty set.

3.2 Mathematical modelling of N-Queens

To investigate the complexity of counting N-Queens and allow SAT based algorithmic implementation, we mathematically model the N-Queens problem. In this section, we provide basic definitions for general N-Queens problems and especially define counting N-Queens problem. We also discuss unsatisfiability of the problem to allow SAT based early detection of unsatisfiability to improve backtracking.

3.2.1 Basic definitions

For the N-Queens problem, we formulate the underlying chessboard as a matrix. We represent a queen of the chessboard with the value **1** in the matrix. So the corresponding matrix for the solution 3.1 is

$$\begin{pmatrix} & & 1 & \\ 1 & & & \\ & & & 1 \\ & 1 & & \end{pmatrix}$$

Due to the constraints of the N-Queens problem, two queens cannot be placed in the same row, column or a diagonal. We use **0** in the matrix to represent that a queen cannot be placed. So the new matrix will be

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

In our implementations for counting N-Queens, we start with an empty board. To allow the SAT way of implementation, we allow an element of the matrix to have $*$ as an additional value to represent an open field. So the initial empty board is

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}$$

Definition 3.2.1 *Let $N \in \mathbb{N}$. An **N-Board** is an $N \times N$ matrix with values $\{0, 1, *\}$.*

An example of a **4-Board** is

$$\begin{pmatrix} 0 & 1 & 0 & * \\ 1 & * & 0 & 0 \\ * & 0 & 1 & 0 \\ 0 & 0 & 0 & * \end{pmatrix}$$

In further discussions, we refer to an N-Board as a **board** i.e., a board with an arbitary size. When discussing a board of specific size, we use the actual representation, for example, 4-Board. We refer to the elements of the matrix (of underlying board representation) as a **field**. We refer $M_{i,j} = *$ as an **open field**. We refer $M_{i,j} = 1$ as a **placed field** and setting an open-field to placed-field is called **placing a queen**. We refer $M_{i,j} = 0$ as a **forbidden field** and setting an open-field to forbidden-field is called **forbidding a field**. The *placing* and *forbidding* a queen allows us to branch on fields resulting in branching trees that allow SAT based implementations.

We consider that the indices of the matrix start from $(1, 1)$, grows from bottom to top and left to right. For example, the ordering of indices for 4-Board is

$$\begin{matrix} (4,1) & (4,2) & (4,3) & (4,4) \\ (3,1) & (3,2) & (3,3) & (3,4) \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (1,1) & (1,2) & (1,3) & (1,4) \end{matrix}$$

There exists a fundamental extension among the boards from the partial assignments of the fields. Let $\phi_B, \phi_{B'}$ be the sets of union of placed and forbidden fields of the corresponding boards B, B'. The boards B, B' have an extension relation denoted by $B \leq B'$ if $\phi_B \subseteq \phi_{B'}$. This fundamental extension between boards creates a partial ordering on the set of all boards. First, we define the set of all N-Boards.

Definition 3.2.2 *For $N \geq 1$, $\mathbb{B}_N$ is the set of all N-Boards and the **set of all boards***

$$\mathbb{B} := \bigcup_{N \in \mathbb{N}} \mathbb{B}_N$$

$\mathbb{B}$ is the union over all $\mathbb{B}_N$ and each $\mathbb{B}_N$ has $3^{(N^2)}$ N-Boards. Now from the fundamental extension relation between boards, there exists a partial order over the set of all boards $\mathbb{B}$. In this order, for every N, there exists one connected component. In each component with N dimension, the empty N-Board is the smallest element, and the boards without open fields are maximal elements. We will use this partial order on the set of all boards and fundamental extension between boards to (later) define unsatisfiability and propagation.

A fundamental invariant of N-Queens problem is that two queens cannot be placed in the same row, column, or diagonal. While it is easy to check if two queens are placed in the same row or column, it is not clear in the case of the diagonals. First, we divide the diagonals into diagonals and anti-diagonals of the board. The diagonals that start from bottom-left to top-right are diagonals. The diagonals that start from bottom-right and end at top-left are anti-diagonals. We provide formal definitions for diagonals and anti-diagonals of a board.

The fields with an equal difference in their coordinates belong to the same diagonal.

Definition 3.2.3 *$M_{i,j}, M_{i',j'}$ are in same **diagonal** if $(i-j) = (i'-j')$.*

We define diagonal as a set of fields on the board.

Definition 3.2.4 *Let $N, M \in \mathbb{N}_0$, $k \in \{1, \ldots, 2 \cdot N - 1\}$. A **diagonal** $\mathbf{d}(\boldsymbol{N}, \boldsymbol{k})$ is the set of fields of the N-Board such that $k = (i-j) + N$ where $M_{i,j}$ is a field of the N-Board.*

For example, the set of fields $\{(2,1), (3,2), (4,3)\}$ is $\mathrm{d}(4,5)$ of a 4-Board:

$$\begin{pmatrix} * & * & \boldsymbol{*} & * \\ * & \boldsymbol{*} & * & * \\ \boldsymbol{*} & * & * & * \\ * & * & * & * \end{pmatrix}$$

Definition 3.2.5 *A diagonal $\mathrm{d}(N,k)$ is the **main-diagonal** of an N-Board if $k = N$.*

For example the set of fields $\{(1,1), (2,2), (3,3), (4,4)\}$ is the *main-diagonal* $\mathrm{d}(4,4)$ of a 4-Board:

$$\begin{pmatrix} * & * & * & \boldsymbol{*} \\ * & * & \boldsymbol{*} & * \\ * & \boldsymbol{*} & * & * \\ \boldsymbol{*} & * & * & * \end{pmatrix}$$

The fields with an equal sum of their coordinates belong to the same anti-diagonal.

Definition 3.2.6 *$M_{i,j}, M_{i',j'}$ are in same **anti-diagonal** if $(i+j) = (i'+j')$.*

We define anti-diagonal as a set of fields on the board.

Definition 3.2.7 *Let $N, M \in \mathbb{N}_0$, $k \in \{1, \ldots, 2 \cdot N - 1\}$. An **anti-diagonal** $\mathbf{ad}(\boldsymbol{N}, \boldsymbol{k})$ is the set of fields of the N-Board such that $k = i + j - 1$ where $M_{i,j}$ is a field of the N-Board.*

For example, the set of fields $\{(4,2),(3,3),(2,4)\}$ is $\mathrm{ad}(4,5)$ of a 4-Board:

$$\begin{pmatrix} * & \boldsymbol{*} & * & * \\ * & * & \boldsymbol{*} & * \\ * & * & * & \boldsymbol{*} \\ * & * & * & * \end{pmatrix}$$

Definition 3.2.8 *An anti-diagonal* $\mathrm{ad}(N,k)$ *is the* ***main-anti-diagonal*** *of an* N*-board if* $k = N$.

For example, the set of fields $\{(4,1),(3,2),(2,3),(1,4)\}$ is the **main-anti-diagonal** $\mathrm{ad}(4,4)$ of a 4-Board:

$$\begin{pmatrix} \boldsymbol{*} & * & * & * \\ * & \boldsymbol{*} & * & * \\ * & * & \boldsymbol{*} & * \\ * & * & * & \boldsymbol{*} \end{pmatrix}$$

If a set of fields do not share the same row, column, diagonal and anti-diagonal, we define them as non-attacking.

Definition 3.2.9 *Let* S *be a set of fields of an* N*-Board.* S *is a set of* ***non-attacking*** *fields if for every* $(i,j),(i',j') \in S$*:*

(i) $i \neq i'$ *and* $j \neq j'$*;*

(ii) $(i-j) \neq (i'-j')$*;*

(iii) $(i+j) \neq (i'+j')$.

Condition (i) specifies that no two fields can share the same row or column. Condition (ii) specifies no two fields can share a same diagonal (see Definition 3.2.3). Condition (iii) specifies no two fields can share a same anti-diagonal (see Definition 3.2.6). For example, the set $\{(1,2),(2,4)\}$ is non-attacking. The fields $(1,2)$ and $(2,4)$ do not share the same row and column. They also do not share the same diagonal or anti-diagonal as the difference, and the sum of their coordinates are not equal.

We now define a partial solution for the N-Queens problem using the definition of non-attacking fields.

Definition 3.2.10 *Let* $N \in \mathbb{N}$ *and* S *be the set of all placed-fields of an* N*-Board.* S *is a* ***partial-solution*** *for* N-Queens *problem if it is non-attacking.*

For example, a partial solution of a 4-Queens problem is

$$\begin{pmatrix} * & * & 1 & * \\ 1 & * & * & * \\ * & * & * & * \\ * & 1 & * & * \end{pmatrix}$$

Here, the set of placed fields $\{(3,1),(1,2),(4,3)\}$ is non-attacking and thus a partial solution. Let $k \in \mathbb{N}_0$, a k-**partial solution** is a partial solution of size

k. The set of placed fields $\{(3,1),(1,2),(4,3)\}$ is a 3-partial solution. We now define the set of all the partial solutions for the N-Queens problem.

Definition 3.2.11 *Let $N \in \mathbb{N}$, $\mathbf{PQ}_{\boldsymbol{N}}$ is the* ***set of all the partial solutions*** *of the* N-Queens *problem and the set of all partial solutions*

$$\mathbf{PQ} := \bigcup_{N \in \mathbb{N}} \mathrm{PQ}_N$$

For example, $\mathrm{PQ}_2 = \{\emptyset, \{(1,1)\}, \{(1,2)\}, \{(2,1)\}, \{(2,2)\}\}$ is the set of all partial solutions for the 2-Queens problem. Let $k \in \mathbb{N}_0$, the set $\mathbf{PQ}_{\boldsymbol{N}}\boldsymbol{(k)}$ is the set of all k-partial solutions. For example, the set of all 1-partial solutions for the 2-Queens problem is $\mathrm{PQ}_2(1) = \{\{(1,1)\}, \{(1,2)\}, \{(2,1)\}, \{(2,2)\}\}$. We extend the definition of a partial solution to a (complete) solution of the N-Queens problem.

Definition 3.2.12 *Let $N \in \mathbb{N}$ and $k \in \mathbb{N}_0$, a k-partial-solution is a (complete)* ***solution*** *of an N-Board if $k = N$.*

For example, the set $\{(3,1),(1,2),(4,3),(2,4)\}$ is a 4-Queens (complete) solution. Its corresponding board representation is

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{3.1}$$

In further discussions, a solution of a board is a complete solution. We define the set of all the N-Queens solutions.

Definition 3.2.13 *Let $N \in \mathbb{N}$, $\mathbf{SQ}_{\boldsymbol{N}}$ is the set of all the solutions for the* N-Queens *problem and the set of all solutions*

$$\mathbf{SQ} := \bigcup_{N \in \mathbb{N}} \mathrm{SQ}_N$$

For example, solution sets of some small N:

1. $\mathrm{SQ}_1 = \{\{(1,1)\}\}$.
2. $\mathrm{SQ}_2 = \mathrm{SQ}(3) = \emptyset$.
3. $\mathrm{SQ}_4 = \{\{(3,1),(1,2),(4,3),(2,4)\}, \{(2,1),(4,2),(1,3),(3,4)\}\}$.
4. $\mathrm{SQ}_6 = \{\{(2,1),(4,2),(6,3),(1,4),(3,5),(5,6)\},$
   $\{(3,1),(6,2),(2,3),(5,4),(1,5),(4,6)\},$
   $\{(4,1),(1,2),(5,3),(2,4),(6,5),(3,6)\},$
   $\{(5,1),(3,2),(1,3),(6,4),(4,5),(2,6)\}\}$.

Among the total number of boards, there are 2^{N^2} boards with no open fields i.e., complete assignments. For each N, the set of all solutions is a subset of these 2^{N^2} board and SQ is the union over all the set of solutions for each N. From the definition of the set of all the solutions for the N-Queens problem SQ_N, we now define the solution count for the N-Queens problem.

Definition 3.2.14 *Let $N \in \mathbb{N}$, the* ***solutions count*** $\#\mathrm{SQ}(N) := |\mathrm{SQ}_N|$.

For example, solution counts for some small N:

1. $\#\mathrm{SQ}(1) = 1$.
2. $\#\mathrm{SQ}(2) = \#\mathrm{SQ}(3) = 0$.
3. $\#\mathrm{SQ}(4) = |\mathrm{SQ}_4| = 2$.
4. $\#\mathrm{SQ}(5) = |\mathrm{SQ}_5| = 10$.
5. $\#\mathrm{SQ}(6) = |\mathrm{SQ}_6| = 4$.

From the basic definitions of the N-Queens problem, we now formally define the counting N-Queens problem in terms of solution count.

Definition 3.2.15 *Given an $N \in \mathbb{N}$, the* ***counting N-Queens*** *problem is to compute $\#\mathrm{SQ}(N)$.*

Using this mathematical modelling of the N-Queens problem, we introduce Queens branching trees (QBT trees) (presented in Section 3.3) to understand the state space and investigate the complexity of the counting N-Queens problem. This same modelling is used in our algorithmic implementation of QBT trees (further discussed in Section 4.2).

3.2.2 Unsatisfiability of N-Queens problem

Understanding the unsatisfiability of a problem is critical in investigating its complexity and for efficient implementations. Particularly in SAT based implementations finding the unsatisfiability earlier improves the solver efficiency. Unsatisfiability conditions naturally arise from the constraints of the problem, and some are more visible than others. It is possible to never run into unsatisfiability with a complete understanding of the problem. In general, we do not have a complete understanding of the problem. From the constraints of the N-Queens problem, however, it is possible to identify unsatisfiability before assigning all the fields of the board. We first define satisfiability of a board in terms of the set of placed fields of the board.

Definition 3.2.16 *An N-Board is a* ***satisfiable*** *board if the set of all the placed fields of the board can be extended to an* N-Queens *solution.*

For example, a 4-Board

$$\begin{pmatrix} * & * & 1 & 0 \\ 1 & * & * & * \\ * & * & 0 & * \\ * & 1 & * & 0 \end{pmatrix}$$

is a satisfiable board. The set of placed fields $\{(3,1),(1,2),(4,3)\}$ can be extended to a solution $\{(3,1),(1,2),(4,3),(2,4)\}$ for 4-Queens problem. A point to be observed is that the set of placed fields of a satisfiable board is also a partial solution. The set of all satisfiable boards naturally arise from the partial order on $\mathbb{B}$ and SQ. The downward-closure of SQ gives the set of all satisfiable boards.

Definition 3.2.17 *An N-Board is a **satisfied** board if the set of all the placed fields is an* N-Queens *solution.*

The board (3.1) is a satisfied board and $\{(3,1),(1,2),(4,3),(2,4)\}$, the set of all placed fields is a 4-Queens solution. The set of satisfied boards is a subset of all the satisfiable boards. Similarly, we define the unsatisfiability of a board in terms of the set of placed fields of the board.

Definition 3.2.18 *An N-Board is an **unsatisfiable** board if the set of all the placed fields of the board can not be extended to a solution (see Definition 3.2.12) of the* N-Queens *problem.*

For example, a 4-Board:

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ 1 & * & * & * \end{pmatrix}$$

is unsatisfiable as the set of placed fields $\{(1,1)\}$ cannot be extended to any 4-Queens solution. This example also shows that a k-partial-solution can be an unsatisfiable board too.

Definition 3.2.19 UB *is the set of **all the unsatisfiable boards**.*

Similar to the set of satisfiable boards, the set of unsatisfiable boards also naturally arise from partial ordering on $\mathbb{B}$. As the set of all satisfiable boards is downward-closed on SQ, the remaining boards are the set of unsatisfiable boards. So the set of unsatisfiable boards is upward-closed, i.e., once a board is unsatisfiable then all the boards that extend from this board are also unsatisfiable. Also, the prime implicants are the minimal unsatisfiable boards.

From the constraints of the N-Queens problem, we define AMO and ALO unsatisfiability.

Definition 3.2.20 *An N-Board is* **AMO** ***unsatisfiable*** *if there exists a row, column, diagonal or an anti-diagonal with more than one field $M_{i,j} = 1$.*

In other words, if a row, column, diagonal or an anti-diagonal of an N-Board has two placed fields then the board is AMO unsatisfiable. We refer this constraint over rows, columns, diagonals, and anti-diagonals as **AtMostOne** constraint. We define $\mathrm{U_{AMO}}$ as the set of such AMO unsatisfiable boards.

Definition 3.2.21 $\mathbf{U_{AMO}}$ *is the set of all* AMO *unsatisfiable boards.*

Simiarly, we define ALO unsatisfiability for an N-Board.

Definition 3.2.22 *An N-Board is* **ALO** ***unsatisfiable*** *if there exists a row or column with every field $M_{i,j} = 0$.*

In other words, in an N-Board, if any row or column has all forbidden fields then the board is ALO unsatisfiable. We refer to the constraints over rows and columns as the **AtLeastOne** constraint. We define $\mathrm{U}_{\mathrm{ALO}}$ as the set of all the ALO unsatisfiable boards.

Definition 3.2.23 $\mathbf{U_{ALO}}$ *is the set of all* ALO *unsatisfiable boards.*

The AMO and ALO unsatisfiable boards are easily identifiable. We define the union of these two set of unsatisfiable boards as $\mathrm{UB}_{\mathrm{AA}}$.

Definition 3.2.24 *The set of* ***all the falsified boards***

$$\mathbf{UB_{AA}} := \mathrm{U}_{\mathrm{AMO}} \cup \mathrm{U}_{\mathrm{ALO}}$$

In our investigations on QBT trees and their implementations, we only need to know whether a board is unsatisfiable. We backtrack if the current node of the tree is unsatisfiable and we define an unsatisfiability test to allow this in the QBT trees. The unsatisfiability test is simply a subset of all the unsatisfiable boards. A node is decided unsatisfiable if it is in the unsatisfiability test.

Definition 3.2.25 *An* ***unsatisfiability test*** U *is a subset of all the unsatisfiable boards* UB*.*

The unsatisfiability test can be the empty set or the complete set of all the unsatisfiable boards. We use the unsatisfiability test as a parameter for defining different classes of trees.

3.2.3 Propagation-Reductions

For efficient implementations, it is possible to find some assignments of fields that maintain the satisfiability and unsatisfiability of the current board (a reduction) while avoiding (some intermediate) unsatisfiable boards.

Definition 3.2.26 *Given an N-Board and an open field $M_{i,j} = *$, a* ***forced assignment*** *is*

1. $M_{i,j} = 1$*, if* $M_{i,j} = 0$ *results in an unsatisfiable board or*
2. $M_{i,j} = 0$*, if* $M_{i,j} = 1$ *results in an unsatisfiable board.*

More generally, forced assignments can be from specific unsatisfiability test. A forced assignment from unsatisfiability test U is simply U-*forced* assignment. Forcing assignments for a board results in a **reduction**. We formulate the propagation-reduction for a board as a pair (B, B'), where the reduced board B' is obtained by forcing some open fields to either placed or forbidden values. We formally define the propagation as a set of propagation-reductions. Remember that boards have a fundamental extension relation $\leq$ creating a partial order over the set $\mathbb{B}$. Now we define the set of all propagation-reductions.

Definition 3.2.27 *For $N \in \mathbb{N}$ the set of all propagation-reductions is*

$$\mathbf{PR}_{\boldsymbol{N}} := \{(B, B') \in \mathbb{B}_N^2 \mid B \leq B' \text{ and } \forall B'' \in \mathrm{SQ}_N : B \leq B'' \Rightarrow B' \leq B''\},$$

while $\mathbf{PR} := \bigcup_{N \in \mathbb{N}} \mathrm{PR}_N$.

In the propagation-reductions in PR, the board B' i.e., the reduced or propagated board of B has more open fields assigned without any change of satisfiability of the board. Similar to unsatisfiability test, we do not (usually) have the set of all propagation-reductions, so we allow subsets of PR. We consider the propagation (i.e. a set of propagation-reductions) as a parameter when defining special classes of trees.

Definition 3.2.28 *The **propagation** P is a subset of all the propagation-reductions* PR.

Further, the minimum level of propagation is no propagation i.e., in a propagation reduction pair (B, B'), both B, B' are the same. So we define the minimum level of propagation P_0 as the set of all propagation-reduction pairs where both the boards are identical.

Definition 3.2.29 *Let $B \in \mathbb{B}$, $\mathbf{P_0}$ is the set of all (B, B) propagation reductions.*

Ideally, we would want PR as the propagation parameter so that we never reach unsatisfiability. Similar to U with the set of all unsatisfiable boards, it is difficult to compute all of the forced assignments (in polynomial time). Instead, *obvious* or easy to find (in polynomial time) propagation-reductions are computed. For example, unit-clause propagation is extensively used in SAT solving, i.e., propagating clauses with a single literal. For our implementations, we consider propagation-reductions that arise directly from the constraints of the N-Queens problem.

Definition 3.2.30 *Let $B, B' \in \mathbb{B}_N$,* AmoAlo *propagation $\mathbf{P}_{\boldsymbol{AA}}$, is the set of all propagation-reductions (B, B') obtained from the* $\mathrm{UB}_{\mathrm{AA}}$*-forced assignments such that B' do not have any* $\mathrm{UB}_{\mathrm{AA}}$*-forced assignments.*

Using the unsatisfiability test and propagation as parameters, we define classes of branching trees in the following section.

3.3 Queens branching trees

The complexity of the counting N-Queens is unclear and considering the Conjecture 2.4.1, we first attempt to understand the complexity of enumerating all the solutions. For theoretical analysis, only underlying classes of recursive trees produced by the algorithms are considered by abstracting away the implementation and hardware details of the algorithms. In particular, a subclass of recursive trees known as *branching* or *splitting* trees are considered. The branching trees allow SAT based implementations, which is our motivation for modelling the N-Queens problem. Remember that in the modelling of the board, each field has three values $\{0, 1, *\}$, to allow branching on an open field by placing and forbidding it.

In this section, we introduce a class of trees QBT, a subclass of recursive branching trees. The QBT trees are similar to the search trees generated by `DLL` counting algorithm (2) but are specific to the N-Queens problem. Instead of boolean formula in CNF representation as an input, an N-Board (see Definition 3.2.1) is given.

A queens branching tree is a full binary tree generated recursively by branching on an open field of the board. The left child is generated by forbidding the open field and right child by placing the open field. The nodes are labelled with propagation-reductions and inner nodes are further labelled with the open field, it is split upon. Here the propagated board in the label of a leaf is either a solution or an unsatisfiable board. We formally define a queens branching tree as:

Definition 3.3.1 *A* ***queens branching tree*** *for size N is a full binary tree, with the nodes labelled by pairs (B, B') in PR_N, and the inner nodes labelled with fields (i, j), such that holds:*

i For the label (B, B') at the root: B is the empty board.

ii For the labels (B, B') of leaves: B' is a solution or B' is unsatisfiable.

iii (i, j) is an open field of B' in (B, B'), and for the left child and its label (B_l, B'_l) holds, that B_l is obtained from B' by forbidding field (i, j), while for the right child and its label (B_r, B'_r) holds, that B_r is obtained from B' by placing field (i, j).

We now define the class QBT as the set of all queens branching trees.

Definition 3.3.2 *The class QBT_N is the set of all queens branching trees of size N and QBT is the union of all QBT_N for $N \in \mathbb{N}$.*

Using the observation that each inner node is split upon one open field, we can provide some (trivial) upper bounds on sizes of queens branching trees.

Lemma 3.3.3 *Let $T \in \mathrm{QBT}_N$, $\#\mathrm{nds}(T) <= 2^{N^2+1} - 1$.*

Proof: In T at each branching level, an inner node is split into two children. As the total number of fields in a board is N^2, the maximum depth for any leaf is N^2. The maximum number of nodes for a full binary tree with depth N^2 is $2^{N^2+1} - 1$. So the total number of nodes of each queens branching tree of size N is atmost $2^{N^2+1} - 1$ irrespective of the branching heuristics used.

From this upper bound, we can see that the state-space of all the possible boards is reduced.

Lemma 3.3.4 *The sizes of trees in QBT_N are significantly smaller than the state space of all the possible N-Boards.*

Proof: In an N-Board, each field has 3 values $\{0, 1, *\}$ resulting in 3^{N^2} possible boards. But from Lemma 3.3.3, the total number of nodes in every queen branching tree is atmost $2^{N^2+1} - 1$. So the class QBT_N already restrict the state space by using the observation that no solution has an open field. Thus our restriction that only open fields are set to either placed or forbidden fields results in a reduction of state-space.

While the class QBT restricts the state-space to some extent, the trees still can be exponentially large and we do not have proper statements on sizes of trees and their possible efficient implementations. So we allow specific unsatisfiability test and propagation as parameters and define special classes of trees. While using the unsatisfiability test U and propagation P for a special class of trees, it is essential that the propagation-reductions P to be unique or at least compatible with U. For example, if there exists propagation-reductions $(B, B'), (B, B'') \in \mathrm{P}$ then (B', B'') must be in U. We call this the **compatibility condition** on U and P or simply U and P are compatible. We now define the subclass of QBT with U and P as additional parameters.

Definition 3.3.5 $\mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$ *is the set of queens branching trees of size N such that for every branching tree holds:*

i For the labels (B, B') of leaves: $B' \in \mathrm{SQ}_N$ or $B' \in \mathrm{U}$.

ii The labels (B, B') of nodes are in P.

iii U *and* P *are compatible.*

and $\mathrm{QBT}(\mathrm{U}, \mathrm{P})$ *is the union of all* $\mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$ *for* $N \in \mathbb{N}$.

In the following section, we define special classes of braching trees with specific unsatisfiability test and propagation.

3.3.1 Special Classes of Branching trees

We first define classes with minimum propagation P_0 and different levels of unsatisfiability test. The most basic subclass of QBT is the class of queen branching trees with empty unsatisfiability test and minimum propagation.

Definition 3.3.6 *The set* $\mathrm{QBT}_N(\emptyset, \mathrm{P}_0)$ *is the class of* ***N-level-zero*** *trees and the class* ***level-zero*** *is the union over all N-level-zero classes for $N \in \mathbb{N}$.*

In the class level-zero, the unsatisfiability test U is an empty set and propagation P_0. Thus every node only terminates when there are no open fields in the board (both boards are same in the propagation-reduction pair). Here a point to be observed is that the size of every N-level-zero tree is $2^{N^2+1} - 1$. This is clear as every leaf is at the depth of N^2 (a similar argument as mentioned in lemma 3.3.3). For example, consider 1-Queens problem, there are 3 possible boards B_0, B_1, B_2 where

1. $B_0 = (*)$ is an empty board,

2. $B_1 = \big(0\big)$ is a falsified board and

3. $B_2 = \big(1\big)$ is a satisfied board.

The Figure 3.3 is the only tree in $\mathrm{QBT}_1(\emptyset, \mathrm{P}_0)$. Also, we only split on an open field, so there is only one unique tree.

$$\begin{tikzcd}
& (B_0, B_0), (1,1) \arrow[dl, dash, "0"'] \arrow[dr, dash, "1"] & \\
(B_1, B_1) & & (B_2, B_2)
\end{tikzcd}$$

Figure 3.3: A 1-level-zero tree.

Here we can see that the number of nodes in a 1-level-zero tree is 3. These trees grow exponentially and from the constraints, it is possible to decide if the board is falsified before assigning all the open fields i.e., strengthening the unsatisfiability test, resulting in a reduction of the tree sizes. We define a subclass of trees with U as the set of all falsified boards $\mathrm{UB}_{\mathrm{AA}}$.

Definition 3.3.7 *The set* $\mathrm{QBT}_N(\mathrm{UB}_{\mathrm{AA}}, \mathrm{PR}_0)$ *is the class of* ***N-level-basic*** *trees and the class* ***level-basic*** *is the union over all the* N*-level-basic classes for* $N \in \mathbb{N}$.

Here the leaves are either falsified or satisfied boards (which are complete assignments). These trees, in theory, can have smaller sizes due to the reduction of state spaces by pruning off the falsified boards. However, it mainly depends on the depth at which falsified boards appear, i.e. if the falsified board appear near to the root, it results in a significant reduction of state space. On the other hand, if the falsified boards do not appear until most of the fields are assigned, the number of nodes will be exponential. Branching heuristics play an essential role in generating smaller trees. Further strengthening the unsatisfiability test, we define a subclass of QBT with the set of all unsatisfiable boards UB as U.

Definition 3.3.8 *The set* $\mathrm{QBT}_N(\mathrm{UB}, \mathrm{P}_0)$ *is the class of* ***N-level-unsat*** *trees and the class* ***level-unsat*** *is the union over all* N*-level-unsat classes for* $N \in \mathbb{N}$.

These trees are powerful as unsatisfiability is immediately identified. The trees being full binary trees, the bounds on the tree sizes can be formulated in terms of the total number of solutions.

Lemma 3.3.9 *Let* $N \notin \{2, 3\}$ *and* $T \in \mathrm{QBT}_N(\mathrm{UB}, \mathrm{P}_0)$*,*

$$(2 \cdot \#\mathrm{SQ} - 1) \leq \#\mathrm{nds}(T) \leq (4 \cdot \#\mathrm{SQ} - 1)$$

Proof: Every inner node of T is a satisfiable node, which is clear as the unsatisfiability is identified by U, terminating the recursion and resulting in a leaf. The parent of each leaf, being a satisfiable inner node, has at-least one satisfied child. So the maximum number of leaves is at most twice the total number of solutions, i.e., #SQ. As these trees are enumerating all the solutions, and every satisfied solution is a complete assignment, the lower bound of leaves is #SQ. Being full binary trees, a tree with k leaves has $2 \cdot k - 1$ total nodes. So lower and upper bounds on $\mathrm{QBT}_N(\mathrm{UB}, \mathrm{P}_0)$ are $(2 * \#\mathrm{SQ} - 1)$ and $(4 \cdot \#\mathrm{SQ} - 1)$ respectively.

Now we define special classes with strengthened propagation. The maximum propagation is **complete propagation** i.e., P = PR, we first define a class of trees with complete propagation.

Definition 3.3.10 *The set* $\mathrm{QBT}_N(\mathrm{UB}, \mathrm{PR})$ *is the class of* N*-**perfect-trees** and the class* ***perfect-trees*** *is the union over all* N*-perfect classes for* $N \in \mathbb{N}$.

From the compatibility condition the U needs to have all the propagated unsatisfiable boards in propagation-reduction pairs from PR. So only UB, the set of all unsatisfiable boards, is compatible with PR when used as a parameter. For the satisfiable problems i.e., $N \notin \{2, 3\}$, the perfect trees never run into unsatisfiability. The tree sizes can be formulated in terms of #SQ as all the leaves here are solutions.

Lemma 3.3.11 *Let* $N \notin \{2, 3\}$ *and* $T \in \mathrm{QBT}_N(\mathrm{UB}, \mathrm{PR})$,

$$\#\mathrm{nds}(T) = 2 \cdot \#\mathrm{SQ} - 1.$$

Proof: As the leaves are exactly the total number of solutions #SQ, being a full binary tree the total number of nodes is $2 \cdot \#\mathrm{SQ} - 1$.

The special classes defined above are either difficult to implement or are exponentially large. For example consider the perfect-trees class, while these trees are powerful, usually, the set of all unsatisfiable boards is not available. Even if available, it is not feasible to compute all the unsatisfiable boards for an efficient algorithmic implementation. A polynomial implementation of either UB or PR will solve the N-Queens problem. However, being an NP-complete problem, there exists no such implementation unless P = NP. So, we define a special class with unsatisfiability test $\mathrm{UB}_{\mathrm{AA}}$ and AmoAlo propagation which is suitable for efficient implementations.

Definition 3.3.12 *The set* $\mathrm{QBT}_N(\mathrm{UB}_{\mathrm{AA}}, \mathrm{P}_{\mathrm{AA}})$ *is the class of* N-AmoAlo-*trees and the class* AmoAlo-*trees is the union over all* N-AmoAlo *trees for* $N \in \mathbb{N}$.

AmoAlo, $\mathrm{QBT}(\mathrm{UB}_{\mathrm{AA}}, \mathrm{P}_{\mathrm{AA}})$, is the main class of trees we investigate. Similar to the level-basic trees (introduced in 3.3.7), we do not have any statements on the tree sizes in AmoAlo. In theory, these trees can have polynomial sizes in output, but we do not have any proof yet. To investigate AmoAlo trees, we provide algorithmic implementation (discussed in Chapter 4) to investigate the tree sizes and their growth with different branching heuristics. In Section 3.4, we discuss tree sizes of different classes and attempt to make some statements. In particular, we make conjectures on AmoAlo trees with different branching heuristics.

3.3.2 Complexity

We use the Random Access Machine (RAM) model (presented in [57]) to compute the complexity of AmoAlo trees and the special classes of QBT in general. As discussed earlier for the trees in different special classes of QBT, at each node, there are 4 main operations:

1. choosing a branching variable using heuristics.
2. propagation.
3. unsatisfiability test.
4. copying of board.

Inside each node, first, the unsatisfiability test is applied to check if the board is unsatisfiable, which takes one time step. If not unsatisfiable, the reduction pair with the current board from the set of propagation-reductions (i.e., a subset of PR) is accessed, which also takes one timestep. Again unsatisfiability test, which takes one timestep, is applied on the propagated board (second board in the reduction pair). If not unsatisfiable, the branching variable is computed using branching heuristics, which also takes one time step. A copy of the current board is created for spawning the second subtree. As the size of the board is N, there are N^2 fields in total and takes N^2 time steps for the total copying. Finally, using the branching variable, two children (the old board and the new copy) are spawn by placing and forbidding the field. The total time steps taken inside each node is $(N^2 + 4)$, so the total time complexity is $\mathrm{O}(\#\mathrm{nds} \cdot N^2)$. Similarly, each node requires N^2 memory units for storing N^2 fields for each board. As the maximum depth of the recursion is N^2 (i.e., the total fields), the space complexity is $\mathrm{O}(N^4)$.

3.4 Research problems

In this section, we discuss our main research problems and state our fundamental conjectures. We want to understand the complexity of the counting N-Queens problem in terms of tree growth and runtime. We focus on QBT and the special classes, especially AmoAlo trees and make conjectures on them. Assuming branching heuristics, propagation and unsatisfiability tests are polynomial in time; the runtime complexity can be investigated in terms of tree sizes. First, we introduce and define some notions on trees that can be used in investigating the complexity of algorithms that produce these trees. For each class, we define a notion maximum tree size in a class of trees.

Definition 3.4.1 *Let* $\mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$ *be some class of trees, the* ***maximum tree size,*** $\mathbf{S}(\boldsymbol{N}, \mathbf{U}, \mathbf{P})$ *is the number of leaves of the tree* $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$ *such that for every* $T' \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$*,* $|\mathrm{lvs}(T)| \geq |\mathrm{lvs}(T')|$.

In other words, maximum tree size in a class of trees is the number of leaves of the tree with maximum number of leaves. We consider the number of leaves as a basic measure rather than nodes directly, as the leaves are essentially the union of unsatisfiable and satisfied boards. We now provide statements regarding maximum tree size for some basic classes of trees.

Lemma 3.4.2 *The maximum tree size of level-basic trees* $\mathrm{S}(N, \emptyset, \emptyset) = 2^{N^2}$.

Proof: From Lemma 3.3.3, the total number of nodes in each level-zero tree is $2^{N^2+1} - 1$. Being full binary trees, the total number of leaves for each tree $\#\mathrm{lvs} = 2^{N^2}$. So the maximum tree size is 2^{N^2}.

Due to the exponential increase in maximum tree size, it is clear that the algorithmic implementation of level-zero trees is impractical. The maximum tree size is not polynomial in the size of output, i.e., the maximum tree size to solutions ratio is exponential. By considering stronger unsatisfiability test, it is possible to provide bounds on the maximum tree size for some classes. We consider the class of trees with the set of all unsatisiifiable boards UB as the unsatisfiability test U.

Lemma 3.4.3 *The maximum tree size of level-unsat trees*

$$\#\mathrm{SQ} \leq \mathrm{S}(N, \mathrm{UB}, \emptyset) \leq 2 \cdot \#\mathrm{SQ}\,.$$

Proof: Let T be a level-unsat tree. From Lemma 3.3.9, for total number of nodes we have

$$(2 \cdot \#\mathrm{SQ} - 1) \leq \#\mathrm{nds}(T) \leq (4 \cdot \#\mathrm{SQ} - 1).$$

Being full binary trees, we get the bounds on total number of leaves as

$$\#\mathrm{SQ} \leq \#\mathrm{lvs}(T) \leq 2 \cdot \#\mathrm{SQ}$$

The bounds on number of leaves also apply on maximum tree size, thus

$$\#\mathrm{SQ} \leq \mathrm{S}(N, \mathrm{UB}, \emptyset) \leq 2 \cdot \#\mathrm{SQ}\,.$$

A point to be noted here is that while it is possible for a level-unsat tree to have the number of leaves $2 \cdot \#\mathrm{SQ}$, it is not clear yet to state that maximum tree size is $2 \cdot \#\mathrm{SQ}$. So for now, we only provide bounds on the maximum tree size for level-unsat trees. In the class of perfect trees, all the leaves are exactly the solutions.

Lemma 3.4.4 *The maximum trees size of class of N-perfect trees*

$$\mathrm{S}(N, \mathrm{UB}, \mathrm{PR}) = \#\mathrm{SQ}\,.$$

Proof: Let T be a perfect tree. From Lemma 3.3.11, we have $\#\text{nds}(T) = 2 \cdot \#\text{SQ} - 1$ and being full binary trees, we have $\#\text{lvs}(T) = \#\text{SQ}$. As every perfect tree has $\#\text{lvs} = \#\text{SQ}$, the maximum tree size is also $\#\text{SQ}$.

While the level-unsat and perfect trees are ideal trees, we do not have algorithms that generate these trees. On the other hand, it is straight forward to implement level-basic trees (i.e., simply branching on an open field), but there are no clear statements (yet) on their tree sizes. While it is possible that there can be some branching heuristics that result in polynomial tree sizes in size of output, we do not know them yet. For investigating these level-basic tree sizes, one can look at the trees produced by different branching heuristics and make conjectures on them. However, directly implementing the exponentially large level-basic trees and trying to find the right branching heuristics that produce output-polynomial tree sizes is impractical. So we implement algorithms that generate AmoAlo trees, a subclass of QBT trees with propagation. Due to propagation, we conjecture that these trees have sizes polynomial in the size of output for every branching heuristics. We provide some conjectures on AmoAlo trees below, and a SAT based framework `GenericSAT` is implemented for generating and investigating these conjectures (discussed in the next Chapter 4).

As we do not have a complete understanding of the solutions, only looking at the leaves in empirical observations might not give a proper insight into the problems. So, we define and use the leaves-to-solutions ratio to discuss conjectures on the sizes AmoAlo trees. Note that this leaves-to-solutions ratio is not confined to AmoAlo trees but can be used in any backtracking trees. We, in fact, will use the leaves-to-solutions ratio for analysing the trees produced by other counting approaches such as #SAT and `Somers'` algorithms in the next chapters.

Definition 3.4.5 *Given a backtracking tree with* $\#\text{lvs}$ *leaves and* $\#\text{SQ}$ *solutions, the* ***leaves-to-solutions ratio*** $\text{q}(N) := \frac{\#\text{lvs}}{\#\text{SQ}}$.

We conjecture that for the AmoAlo trees, the leaves-to-solutions ratio is some polynomial in the size of input N.

Conjecture 3.4.6 *For* $\text{QBT}_N(\text{UB}_{\text{AA}}, \text{P}_{\text{AA}})$ *i.e., the* N*-AmoAlo trees, there is a* $k \in \mathbb{N}_0$ *such that*

$$\text{q}(N) = \text{O}(N^k).$$

The tree with a leaves-to-solutions ratio that is polynomial in the size of input has tree size polynomial in output (i.e., number of solutions). We define the notion output-polynomial for a tree, i.e., the tree size is polynomial in the size of the output. Here, we define the output-polynomial notion for a tree in terms of the leaves-to-solutions ratio.

Definition 3.4.7 *Let* $N \in \mathbb{N}$*,* $k \in \mathbb{N}_0$ *and* $T \in \text{QBT}_N$*.* T *is said to be* ***output-polynomial*** *in size if the leaves-to-solutions ratio* $\text{q}(N) = \text{O}(N^k)$.

We use the output-polynomial notion for trees produced by different approaches and is not confined to QBT trees. From Conjecture 3.4.6, the AmoAlo trees have a polynomial $\mathrm{q}(N)$, so the algorithms that produce AmoAlo trees for enumerating all the solutions have output-polynomial run time for every branching heuristics. Now assuming the conjecture for AmoAlo tree sizes is correct, we make further conjectures on AmoAlo trees with different branching heuristics. Implementations of these branching heuristics are provided in the next Chapter. First, we provide conjectures for static ordering heuristics (discussed in detail in Subsection 4.2.3) in AmoAlo trees. Among the static ordering heuristics we conjecture that the *FirstRow* heuristics has constant leaves-to-solutions ratio.

Conjecture 3.4.8 AmoAlo *trees with FirstRow heuristics have* $\mathrm{q}(N) = \mathrm{O}(1)$*, and close to* 8.2*.*

We conjecture that this constant factor is due the choice of fields in these ordering avoiding many exponential unsatisfiable subtrees. Further, we conjecture that the *SquareEnum* heuristics have leaves-to-solutions ratio linear in N.

Conjecture 3.4.9 AmoAlo *trees with SquareEnum heuristics have* $\mathrm{q}(N) = \mathrm{O}(N)$*, and close to* $-44.09 + N \cdot (6.6715)$*.*

We conjecture that this relatively large $\mathrm{q}(N)$ is due to presence of the N-Rooks unsatisfiable cases (further discussion in Section 6.1). On the other hand, in dynamic ordering heuristics, we conjecture that the *Taw* heuristics have a constant factor leave-to-solutions ratio that is better than the rest of the heuristics.

Conjecture 3.4.10 AmoAlo *trees with Taw heuristics have* $\mathrm{q}(N) = \mathrm{O}(1)$*, and close to* 2.3*.*

Also, we conjecture that even the *Antitaw* heuristics which seems to produce the largest AmoAlo trees have a polynomial leaves-to-solutions ratio.

Conjecture 3.4.11 AmoAlo *trees with Antitaw heuristics have leaves/solutions ratio* $\mathrm{q}(N) = \mathrm{O}(N^k)$*, and* k *close to* 4*.*

While $\mathrm{q}(N)$ gives an insight into the average distribution of solutions among all the leaves, it does not provide information regarding the waiting time or **delay** between each solution. In other words, there can be exponential unsatisfiable leaves between two consecutive satisfiable leaves, thus resulting in exponential waiting time for the next solution. Even though the q is relatively small, as the solutions grow exponentially, the unsatisfiable leaves can occur consecutively leading to exponential waiting time for the next solution. To investigate the gap between consecutive satisfied solutions, we define a notion **maximum unsat distance** as the largest number of consecutive unsatisfiable leaves for a tree.

Definition 3.4.12 *Let* $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$*,* ***maximum sat gap****,* $\mathrm{maxsgap}(T)$ *is the maximum number of consecutive falsified leaves in the inorder-traversal in* T*.*

In inorder-traversal, we consider the maximum number of consecutive falsifed leaves. Now observing maxsgap(T) will provide an insight into waiting time between two consecutive solutions (satisfied leaves). This notation, in fact, directly relates to **polynomial-delay** as used in complexity theory. That is, if the waiting time between two solutions is some polynomial, then the algorithm has polynomial-delay.

Definition 3.4.13 *Given a $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$ tree, T has* ***polynomial delay*** *if the maximum sat gap,* $\mathrm{maxsgap}(T) = \mathrm{O}(N^k)$.

Instead of directly using maxsgap, we use a more natural notation related to exponential unsatisfiable subtrees which is closely related to the NP-complete completion of the N-Queens problem. First, an **unsatisfiable subtree** of a tree is the subtree with all the nodes unsatisfiable.

Lemma 3.4.14 *Let $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$, if a node of T is an unsatisfiable node then all its children are also unsatisfiable.*

Proof: Let B, B' be a parent and its child respectively of T, and assume that the parent is unsatisfiable whereas the child is satisfiable. From the fundamental extension on boards and partial ordering on the set of boards, it is clear that a child node is simply an extension of the parent node. So $B \leq B'$, that implies $\phi_B \subseteq \phi'_B$ the corresponding partial assignments (representations). However, we know that, if a partial assignment ϕ is satisfiable then every partial assignment ϕ' that is a subset must also be satisfiable. The condition that the parent must be satisfiable contradicts our assumption that the parent is unsatisfiable, so every child of an unsatisfiable node is also unsatisfiable.

Now we define, **maximum unsatisfiable subtree** as the unsatisfiable subtree with the maximum number of leaves of a tree. That is, if there are some k unsatisfiable subtrees in a tree T, then the unsatisfiable subtree with the maximum number of leaves is maximum unsatisfiable subtree. Now using these definitions, we define the notation **size of maximum unsatisfiable subtree** as the number of leaves in the unsatisfiable subtrees with the maximum number of leaves.

Definition 3.4.15 *Given a tree $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$,* ***size of maximum unsatisfiable subtree****,* $\mathrm{maxusat}(T)$ *is the total leaves of the maximum unsatisfiable subtree of T. If there exists no unsatisfiable subtree in T, then* $\mathrm{maxusat}(T) = 0$.

There exists a fundamental constraint on the notions maxsgap and maxusat i.e., maxusat is always less than or equal to maxsgap.

Lemma 3.4.16 *Given a tree $T \in \mathrm{QBT}_N(\mathrm{U}, \mathrm{P})$,* $\mathrm{maxusat}(T) \leq \mathrm{maxsgap}(T)$.

Proof: Let $\mathrm{maxusat}(T) > \mathrm{maxsgap}(T)$. In other words, we assume that there exists an unsatisfiable subtree in T with the number of leaves greater than maxsgap(T). However, in inorder traversal, the leaves of the unsatisfiable subtree are consecutive. Thus, the number of consecutive unsatisfiable leaves is at least maxusat(T). That implies the maximum number of consecutive unsatisfiable leaves maxsgap(T) is at least maxusat(T), which is a contradiction to our assumption. So, maxusat$(T) \leq$ maxsgap(T).

We use *size of maximum unsatisfiable subtree*, maxusat as a lower bound to show the super-polynomial delay in the counting N-Queens problem. The reason we consider maxusat is that it nicely captures the NP-completeness of the completion version of the N-Queens problem. The NP-complete completion problems have some instances which result in exponential unsatisfiable subtrees. So, we conjecture that maxusat(T) is exponential indicating super polynomial delay in AmoAlo trees.

Conjecture 3.4.17 *Let* $T \in \mathrm{QBT}_N(\mathrm{UB}_{\mathrm{AA}}, \mathrm{P}_{\mathrm{AA}})$, $\mathrm{maxusat}(T) = \Omega(k^N)$ *where* $k > 1$.

Even the best available heuristics do not seem to avoid the hard instances strengthening the Conjecture 3.4.17. The experimental setup for investigating the conjectures on AmoAlo trees is discussed in the following chapter. The conjectures are strengthened by our heuristic argumentation provided in Section 4.3.

Chapter 4

Investigating AMOALO trees

In this chapter, we introduce and expand our `GenericSAT`, a SAT based framework for investigating the complexity of AmoAlo trees introduced in Chapter 3. While the focus of `GenericSAT` is more on analysing sizes of AmoAlo trees and other complexity questions, we also provide an efficient implementation for counting solutions. Here we look at the growth of trees produced by different heuristics to investigate if all the trees are output-polynomial in size. Further, we consider the presence of exponential unsatisfiable subtrees to answer polynomial delay question. In Section 4.1, we briefly discuss observations of our initial investigations of the counting N-Queens problem. Further, we provide our motivation and design decisions for the `GenericSAT` framework. In Section 4.2, the implementation of software and explanation of branching heuristics is provided. In Section 4.3, heuristic argumentation is provided for strengthening conjectures on AmoAlo trees.

4.1 Initial investigations

From the existing literature, backtracking strengthened with symmetry breaking and hardware acceleration seems to be the state-of-the-art (discussed briefly in Subsection 5.1.2) for the N-Queens problem. `Somers'` algorithm (presented in Subsection 5.1.1) is a simple, but optimised bitwise implementation that performs well for $N \leq 23$ and shows the efficiency of a simple backtracking approach. Due to the exponential increase in the state space, simple backtracking is not scalable despite its efficiency. On the other hand, SAT solvers seem to generate smaller search trees, possibly due to propagation. While SAT based implementations have the same worst-case complexity as the other backtracking algorithms, they tend to perform exponentially better in most practical problems. Interestingly `tawSolver` (more discussion in Subsection 4.2.4), a simple `DLL` solver seems to perform better than advanced `CDCL` solvers for example `sharpSAT` (briefly discussed in Subsection 5.2.2). From these observations, we conjecture that advanced techniques such as preprocessing, component caching and implicit boolean clause propagation (IBCP) (discussed in Subsection 5.2.2) do not improve the counting N-Queens significantly (presented in Conjecture 2.4.2). So we implement a backtracking framework with propagation and branching heuristics specific for chessboard related problems especially for the N-Queens problem but without the overhead of handling clauses as in general SAT implementations.

4.2 GenericSAT

`GenericSAT` is a SAT based framework developed in C++ with efficient propagation and specialised branching heuristics. `GenericSAT` is an open source software and is available in `oklibrary`[1]. The underlying backtracking framework is similar to that of the `DLL` algorithms (as presented in Subsection 2.4.1). The major difference is that we use board (see Definition 3.2.1) as the underlying datastructure for efficient propagation (further discussed in Subsection 4.2.1) instead of input in CNF representation.

Algorithm 3 Backtracking in GenericSAT

```
    Input: Active clause-set ACLS
    Output: Returns statistics including solution count
 1: function BACKTRACK
 2:     if ACLS is satisfied then return sbstats
 3:     if ACLS is falsified then return fbstats
 4: Branching with heuristics:
 5:     bv ← heuristics(ACLS)
 6:     copy ACLS → ACLS′
 7: Left subtree:
 8:     lstats ← ACLS′.set(bv, $\bot$)
 9: Right subtree:
10:     rstats ← ACLS.set(bv, $\top$)
11:     return lstats + rstats
```

Algorithm 3 presents an outline of the underlying backtracking algorithm in `GenericSAT`. We recursively split on the Active clause-sets (see Subsection 4.2.1) which are essentially boards with propagation into two subtrees (resulting in full binary trees). In each call of the recursive algorithm, if the clause-set is already satisfied (in the Line 2) statistics for a satisfied board are returned. If it is falsified, (in the Line 3) statistics for a falsified board are returned. A branching variable, i.e., an open field in the board is chosen (in the Line 5) by some branching heuristics. A copy of clauseset is generated so that in one clauseset the branching variable is set to $\bot$ and $\top$ in another. The AmoAlo propagation occurs inside the set function (in the Lines $8, 10$). A point to be noted is that we also check if the clauseset is falsified after propagation in the set function. Finally, (in the Line 11), the summation of statistics from both subtrees are returned.

Active clause-sets and branching heuristics are two essential implementation aspects of `GenericSAT` for investigation of tree sizes. In the following sections,

[1] https://github.com/OKullmann/oklibrary

we provide a brief explanation of the Active clause-sets and give implementation details for branching heuristics.

4.2.1 Active clause-sets

To utilise clause propagation from SAT solving and to take advantage of the structure of the chessboard related problems, we use Active clause-sets. They behave similar to clauses, but the propagation can be done efficiently. These Active clause-sets are essentially matrices representing the underlying chessboard of the N-Queens problem with propagation. For the implementation, we use the mathematical modelling of the N-Queens problem that we discussed in Section 3.2. The AmoAlo propagation algorithmically implemented in `GenericSAT`.

The N-Queens problem, when formulated in CNF, has many binary clauses due to AtMostOne constraints. Placing a queen (i.e. setting a variable to true) results in many new unit-clauses and more propagation. However, handling the propagation through unit-clauses involves much overhead and can be avoided by taking advantage of the structure of the chessboard problems. In Active clause-sets, we propagate through rows, columns, diagonals, and anti-diagonals of the underlying board. Thus efficient implementation and optimisations can be made over loops. We consider AmoAlo propagation to implement AmoAlo trees. Here we provide example for propagation, consider a 4-**Board** (see Definition 3.2.1):

$$\begin{pmatrix} \mathbf{1} & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}$$

Setting field $(1, 1)$ to 1 results in AMO-forced assignments. To simulate the propagation in clause-sets, we set all the open fields in the same row, column and diagonals to 0. The resulting board will be

$$\begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & * & * \\ \mathbf{0} & * & \mathbf{0} & * \\ \mathbf{0} & * & * & \mathbf{0} \end{pmatrix}$$

For further efficiency we maintain *ranks* for each row, column, diagonal and anti-diagonal i.e. we track the number of open, placed and forbidden fields in each of them. For every change in the state of a field, we update the ranks for corresponding row, column, diagonal and anti-diagonal. We stop propagation if the rank for open fields becomes 0, i.e., if there are no more open fields, thus increasing the efficiency of implementation. Similarly, we also implement the ALO-forced assignments by simply placing the only open field where every other field is forbidden in a row or column. By applying AMO and ALO forced assignments iteratively we simulate the AmoAlo propagation.

Active clause-sets are powerful data structures for chessboard related problems as they preserve the state of the board while allowing efficient propagation.

This preservation of the state of the board allows polynomial implementations of unsatisfiability tests directly on the board.

4.2.2 Branching heuristics

In QBT trees, the branching heuristics affect the tree size significantly. In this Section, we briefly provide an overview of branching heuristics that are implemented in `GenericSAT` for counting N-Queens. Detailed discussion on the theory behind branching heuristics is provided in [36]. We consider both fixed ordering and dynamic ordering heuristics. In fixed ordering heuristics we consider *FirstRow* and *SquareEnum* heuristics. We choose the first open field in the specified order of fields. We consider *Taw* heuristics and *Antitaw* heuristics, which are weight based dynamic heuristics.

Among the fixed ordering heuristics, the *FirstRow* ordering (will be discussed in the next Section) is the classical (or default) ordering for solving the N-Queens problem. For example in `Somers'` algorithm (discussed in Subsection 5.1.1), the queen is set to be placed (by flipping the bit) row-wise from the bottom to top and column-wise left to right which is same as *FirstRow* enumeration. However, due to the restricted (but optimised) bitwise implementation, it is not possible to choose fields in a different order in the `Somers'` algorithm. However, we overcome this difficulty using SAT based implementation that allows choosing any arbitrary field and branch to generate two subtrees. This greater flexibility allows us choosing the fields such that the propagation can be maximised.

In the following sections, we expand each of these heuristics and provide their pseudocode. As we want to show that AmoAlo trees are polynomial in size of output with every branching heuristics, it is essential to consider heuristics that produce larger trees. The heuristics *SquareEnum* and *Antitaw* heuristics (see Subsubsection 4.2.4.2) are investigated which seems to produce larger trees.

4.2.3 Static ordering: FirstOpen

We divide the implementation of FirstOpen heuristics into two steps. The first step is to generate the order of the fields and provide it as an input for heuristics function (along with the board). In the second step, the first open field is chosen for the branching from the ordered fields. The second step is common for every FirstOpen heuristics, and we first provide the (simple) algorithm for its implementation.

In the algorithm 4, the ordered vector of fields and the N-Board is the input for the branching function. In Lines, 2 and 3, each field of the board is checked if it is open in the order of the fields provided in the input vector. We return the open field if found, or return $(0, 0)$ if no open field exists. In the implementation, the first step, i.e. enumerating fields, differ for each heuristic. We briefly discuss each of the implementations and provide the algorithms in the following sections.

Algorithm 4 FirstOpen

```
    Input: Ordered vector of fields vec, Board $B$
    Output: first open field $(i, j)$
1: function FIRSTOPEN
2:     for each field $(i, j)$ in vec do
3:         if $B_{i,j} = *$ then return $(i, j)$
4:     return $(0, 0)$
```

4.2.3.1 FirstRow

In the *FirstRow* enumeration, we enumerate the fields of the N-Board row-wise from bottom to top and left to right. As mentioned in the previous Section, *FirstRow* enumeration is used in classical backtracking for solving the N-Queens problem. Here the enumeration can be formulated as minimising the indices of the N-Board (i, j), first minimising i and then the j. For example, $\{(1,1),(1,2),(2,1),(2,2)\}$ is enumeration order for 2-Board. To understand the enumeration better we provide the order of enumeration of fields on a 4-board as an additional example:

$$\begin{matrix} 13 & 14 & 15 & 16 \\ 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \end{matrix}$$

In the algorithm 5, the size of the board N is the input. In Lines 2 to 4, we loop through 1 to N, for i and j, to generate the order of fields and append to the vector. In the Line 5, we return the vector of the ordered fields.

Algorithm 5 FirstRowEnum

```
    Input: $N$
    Output: Row wise enumeration vector of fields: vec
1: function FIRSTROWENUM
2:     for $i$ from 1 to $N$ do
3:         for $j$ from 1 to $N$ do
4:             append $(i, j)$ to vec
5:     return vec
```

4.2.3.2 SquareEnum

In *SquareEnum* enumeration, we iteratively choose the fields which are in some x distance from $(0,0)$ (bottom left point, as a starting point) increasing one unit at a time. Here the distance of field (x, y) from $(0,0)$ is simply the minimum value of x and y. We first choose $(1,1)$ which has a distance of 1 from $(0,0)$, then we choose fields $\{(2,1),(2,2),(1,2)\}$ which are 2 units away from $(0,0)$. This choosing, in a sense, is *sweeping* the fields radially (from left to right) increasing distance one unit after each sweep. For example, $\{(1,1),(2,1),(2,2),(1,2)\}$ is square enumeration order for 2-Board. To visualise the choosing of fields, here is an example ordering for the 4-Board:

10	11	12	13
5	6	7	14
2	3	8	15
1	4	9	16

The Algorithm 6, presents the implementation of *SquareEnum* enumeration of the fields. In the Line 2, we loop through each distance unit starting from 1. In the Lines 3 to 6, we first choose (and append to vec) the fields that are in i distance from $(0, 0)$. It can be observed that first fields with distance i as row index are chosen and then the fields with i as a column index. In the Line, 7, enumeration vector vec is returned.

Algorithm 6 SquareEnum

```
    Input: $N$
    Output: Square enumeration vector of fields: vec
1: function SQUAREENUM
2:     for each $i$ in range of $1 \cdots N$ do
3:         for $j$ in range of $(1 \cdots i)$ do
4:             append $(i, j)$ to vec
5:         for $j$ in range of $(i \cdots 1)$ do
6:             append $(j, i)$ to vec
7:     return vec
```

4.2.4 Dynamic ordering

Dynamic ordering of variables instead of static ordering allows more flexibility to choose better branching variables by considering the previous progress. In our initial investigations, `tawSolver` seem to generate smaller trees for the N-Queens problem. The `tawSolver` (see Paper [1] for implementation details) is a simple `DLL` solver with unit-clause propagation and branching heuristics with particular focus on van der Waerden problems. Being a `DLL` algorithm, it has a similar structure as detailed in Subsection 2.4.1. It is a *zero-look-ahead* solver, in the sense that the heuristics choose branching variables without assigning them (i.e., not looking ahead) but by estimating the future progress using some measures. We simulate the `tawSolver` heuristics in our `GenericSAT` framework to reproduce the smaller node count.

To compute the *measure* (in a sense the anticipated progress) of a variable, the lengths of clauses that the variable appear are considered, and tuples are generated depending on the number of branches. Now the measures (of progress) of different variables are compared, and a branching variable is chosen either by maximising or minimising the measure by a projection function. Kullmann [37] provides detailed explanation on the theory of branching heuristics and discusses the computation of measures on tuples. As we want to investigate branching heuristics with both smaller trees and larger trees, we implement heuristics both maximising and minimising the measure anticipating propagation (the projecting function used in `tawSolver`). We simulate the measuring function on variables (fields in our implementation) in `GenericSAT` by considering the *ranks* of fields, i.e. the number of open, placed and forbidden fields in the corresponding row, column, and diagonals (encoding the lengths of clauses of the corresponding CNF representation). The basic idea behind the estimation of measure is to estimate future propagation. As discussed earlier, placing a field results in AMO propagation, i.e., all the open fields in the same row, column, diagonal and anti-diagonal are forbidden. So by looking at `odegree`, i.e. the number of open fields in the same row, column, diagonal and anti-diagonal of a field gives the number of fields that will be propagated. Similarly, the number of open fields in the same row and column are considered to estimate the propagation when the field is forbidden. Weights are considered for prioritising some clause lengths. For example, unit-clause propagation might result in a further AMO propagation, so it is essential to prioritise the clauses with length 2. The computed weighted measures of all the open fields are either maximised or minimised depending on the heuristics. Here we use two projection functions product-rule and sum-rule. The product-rule multiplies the two measures (one from placing the field and another from forbidding it) for each variable whereas sum-rule adds them.

4.2.4.1 Taw heuristics

We implement the exact *Taw* heuristics to simulate the `tawSolver` node count. Here we maximise the projections obtained by product-rule function and then sum-rule on open fields to select the best branching field for smaller trees.

The Algorithm 7 presents the implementation of the *Taw* heuristics. It takes board (presented in Definition 3.2.1) as an input and returns a branching variable. Functions `neglitw` and `poslitw` gives measures on each open field whereas the function `weight` returns the specified weight for a clause length. Function `heuristics` returns a tuple of weighted heuristics, which are the measures computed on the open field. The Lines 11 and 13 presents the product-rule and sum-rule projection functions. In Lines 6 to 8, we initialise the maxp, maxs and branching variable bv. In the Lines from 9 to 17, we try to maximise first the product projection and then (in case of ambiguity) sum projection to choose a branching field among the open fields.

4.2.4.2 Antitaw heuristics

The functions for computing measures such as `heuristics`, `neglitw`, `poslitw` and `weight` are exactly the same as *Taw* heuristics. The only difference is that we try to minimise the projections from product-rule and sum-rule (instead of maximising).

The Algorithm 8 is the implementation of the *Antitaw* heuristics. In Lines 6 to 8, we initialise the minp, mins and branching variable bv. In lines from 5 to 13, we minimise the projections from product-rule and sum-rule. Choosing the

Algorithm 7 Taw Heuristics

```
    Input: Board board
    Output: Branching variable bv
function WEIGHT(cl) return weight of cl
function NEGLITW(x) return (odegree($v$) $*$ weight(2))
function POSLITW(x) return (weight(or_rank($x$)) + weight(oc_rank($x$)))
function HEURISTICS(v) return {neglitw($v$), poslitw($v$)}
function TAWHEURISTICS
    maxp := 0
    maxs := 0
    bv ← {0, 0}
    for each open field vec in board do
        $x, y \leftarrow$ heuristics($v$)
        prod ← $x \times y$
        if prod < maxp then continue
        sum ← $x + y$
        if prod > maxp then maxp ← prod
        else if sum ≤ maxs then continue
        maxs ← sum
        bv ← $v$
    return bv
```

Algorithm 8 *Antitaw* Heuristics

```
    Input: Board board
    Output: Branching variable bv
function Antitaw HEURISTICS
    minp := ∞
    mins := ∞
    bv ← {0, 0}
    for each open field vec in board do
        $x, y \leftarrow$ heuristics($v$)
        prod ← $(x \times y)$
        if prod > minp then continue
        sum ← $(x + y)$
        if prod < minp then minp ← prod
        else if sum ≥ mins then continue
        mins ← sum
        bv ← $v$
    return bv
```

Heuristics	**q**	**maxusat**	**qt**
Taw	≈ 2.32	$\mathrm{O}(1.55^N)$	$\approx 4.63e-06$
FirstRow	≈ 8.17	$\mathrm{O}(1.98^N)$	$\mathrm{O}(N)$
SquareEnum	$\mathrm{O}(N)$	$\mathrm{O}(4.26^N)$	$\mathrm{O}(N^{2.28})$
Antitaw	$\mathrm{O}(N^{4.94})$	$\mathrm{O}(4.05^N)$	$\mathrm{O}(N^{5.47})$

Figure 4.1: `GenericSAT` heuristic argumentation with different branching heuristics.

branching fields in this manner postpones the propagation as much as possible, thus resulting in larger trees as the unsatisfiability is not found until more fields are assigned.

4.3 Heuristic argumentation and discussion

The main research questions are about the sizes of the AmoAlo trees, which indirectly show output-polynomial runtime of the algorithms that produce the trees. As discussed in Section 3.4, the value of leaves-to-solutions ratio $q(N)$ is used to analyse the tree sizes. In previous sections, we discussed the implementation of AmoAlo trees in the `GenericSAT` software. Using the different branching heuristics (as explained above), we generated data for the AmoAlo trees. The data files for each of these branching heuristics are available in `oklibrary`. Instructions for data generation are provided in each file so that the results can be reproduced. We generated data for the N-Queens problems with $N \leq 20$ and only up to $N = 18$ for *Antitaw* heuristics as it already took 40 days of computation for $N = 18$. For each branching heuristics, we then modelled leaves-to-solutions ratio, sizes of maximum unsatisfiable sub-trees and runtime-to-solutions ratio to investigate the tree growth in these trees.

The Table 4.1, provides the data for different branching heuristics for the N-Queens problem. The q in the data represents the leaves-to-solutions ratio, as discussed earlier. Similarly, maxusat is the number of leaves in the maximum unsatisfiable subtrees. The data qt is the runtime-to-solutions ratio for each heuristic. From the data, it is clear that AmoAlo trees with different branching heuristics have q leaves-to-solutions ratio at most in N^k, where k is some constant. These observations strengthen our conjecture that AmoAlo trees with every branching heuristics are output-polynomial in size. Further AmoAlo trees with *Taw* heuristics have a constant factor of $q = 2.3$ and seem to decrease with an increase in N (see Figure 4.2). The branching heuristics *FirstRow* has the leaves-to-solutions ratio q as a constant factor of 8.2. Remember that the underlying enumeration is the same as the classical `Somers'` algorithm (detailed discussion in Subsection 5.1.1). The only difference is that the AmoAlo trees have additional ALO propagation. While the q for `Somers'` algorithm (see Subsection 5.1.1) is linear in N due to additional ALO propagation it reduced to a constant factor. Here the underlying reason is that the work done at each node in the `Somers'` algorithms is linear in N, whereas linear in N^2 in `GenericSAT` implementation. Coming to heuristics that produce large

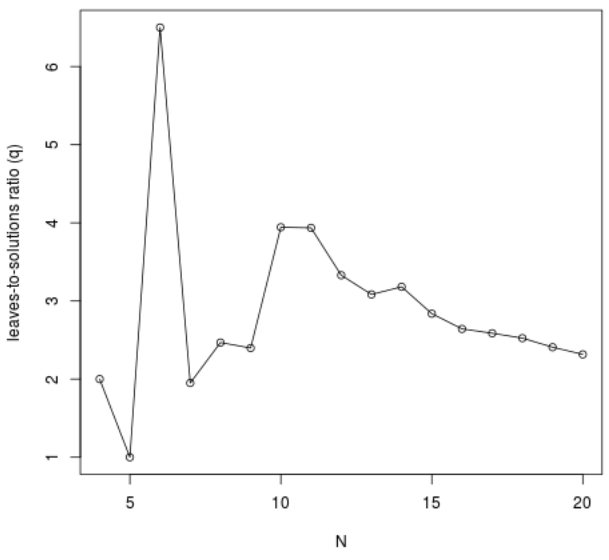

Figure 4.2: Leaves-to-solutions ratio for *Taw* heuristics.

trees, *SquareEnum* heuristics has a linear factor $q \approx -44.0872 + N \cdot (6.6715)$. Whereas the *Antitaw* heuristics has a polynomial factor $q(N) \approx 0.001 \cdot N^{4.94}$. Even though the leaves-to-solutions ratio here is much worse than the other heuristics which are constant, the q is still polynomial in input N.

Now we consider the question of having a polynomial-delay in AmoAlo trees. From the table 4.1, it is clear that the size of maximum unsatisfiable subtrees maxusat is exponential for all 4 branching heuristics. While some have smaller coefficients than other, unsatisfiable subtrees seems to grow exponentially for all the heuristics. The presence of exponential unsatisfiable subtrees strengthen our conjecture that AmoAlo trees do not have a polynomial delay (see Conjecture 3.4.17). Looking at the runtime-to-solutions ratio qt, it seems to be constant for *Taw* heuristics. However, from our complexity analysis, we computed that the runtime is $\mathrm{O}(\#\mathrm{nds} \cdot N^2)$ so the runtime-to-nodes will be $\mathrm{O}(N^2)$. This difference is due to the modelling for small N, with larger N the qt might change significantly. While the qt ratio for *SquareEnum* and *Antitaw* seems to be worse (which is intuitive), they are still polynomial in size of input N. This heuristic argumentation strengthens our conjectures on the sizes of trees and thus indicating the possibility of output-polynomial runtime algorithms for the counting N-Queens problem.

Chapter 5

Counting N-Queens with the state-of-the-art solvers

In the previous chapter, we investigated AmoAlo trees in detail and provided empirical evidence to strengthen our conjectures. In this chapter, we discuss other approaches and their performance on the N-Queens problem. We first discuss specialised backtracking algorithms, which are the current state-of-the-art for the counting N-Queens. The #CSP algorithms produce similar search trees as ours, we investigate if these trees also have output-polynomiality. We discuss the surprisingly *bad* performance of `sharpSAT` solver on counting N-Queens contrary to N-Rooks problem. We also compare the sizes of d-DNNF (see Subsection 5.4.1) trees produced by knowledge compilers with our AmoAlo trees.

5.1 Specialised solvers

Specialised solvers are usually simple backtracking algorithms with symmetry breaking and hardware acceleration. In this section, we first look at the efficient `Somers'` algorithm and its implementation in detail. It speeds up the bit implementation for its performance and using only vertical symmetry to reduce the computation into half. Further, we briefly discuss the specific implementations the counting N-Queens problems for $N \geq 24$.

5.1.1 Somers' algorithm

The basic idea behind `Somers'` algorithm (available in `oklibrary`) is speeding up the backtracking of the N-Queens problem by using bit-level operations. The `Somers'` algorithm is a simple and efficient algorithm which performs well for $N \leq 23$. In this section, we provide the pseudocode of the algorithm and further discuss the implementation. We also provide an analysis of the performance of the algorithm for counting all solutions.

Pseudo code of `Somers'` algorithm is presented in Algorithm 9. At each level, the state of the backtracking is represented by three 32 (or 64, can be choosen) bit words. `Column` (col) is the placements of current queens (each placed column bit is set to 1). `Forbidden diagonals` (fd) represents forbid-

Algorithm 9 `Somers'` algorithm

```
    Input: Bit arrays representing state of the board: avail, col, fd, fad, size
    Output: Solution count
 1: function SETBIT(x, i)
 2:     $x[i] \leftarrow 1$ return x
 3: function SETRIGHTBIT(x, i)
 4:     if $(i+1)$ is not $N$ then $x[i+1] \leftarrow 1$
        return x
 5: function SETLEFTBIT(x, i)
 6:     if $i$ is not 0 then $x[i-1] \leftarrow 1$
        return x
 7: function BACKTRACKING
 8:     if size +1 is equal to $N$ then
 9:         count ← count +1
10:         return count
11:     Right shift fd
12:     Left shift fad
13:     for each open position $i$ in avail do
14:         ncol ← setbit($col, i$)
15:         nfd ← setrightbit($fd, i$)
16:         nfad ← setleftbit($fad, i$)
17:         navail ← ¬(ncol | nfd | nfad)
18:         if navail then backtracking(navail, ncol, nfd, nfad, size +1)
```

den columns due to diagonal constraints. `Forbidden anti-diagonals` (fad) represents forbidden columns due to anti-diagonal constraints. size represents the number of queens set. `Available` (avail) represents columns available excluding forbidden fields due to (anti)diagonals in the particular call. Function `setbit` sets a specific bit whereas functions `setrightbit` and `setleftbit` set right and left bits respectively. `Somers'` algorithm takes avail, col, fd, fad and size as input. In the Line 8, if the current row is the last row we increment the solution counter (i.e., a solution is found). In Lines 9 and 10, we rightshift and leftshift fd and fad respectively. In Lines 12 to 15, we update avail, col, fd and fad by forbidden fields due to new queen placement. In Line 16, if there is at least one open bit in navail, then we call backtracking function with updated bit arrays.

5.1.1.1 Experimental analysis

The `Somers'` algorithm performs well for $N \leq 20$ and scales up to $N = 23$ using parallelisation, however, the runtime increases exponentially. Moreover, the runtime for $N \leq 16$ is too small resulting in noise whereas $N \geq 21$ already takes few days (increasing exponentially). Thus we are unable to model the qt accurately for comparison. The data is available in N-Queens-data[1)]. A simple parallel implementation using futures (available in `oklibrary`) is developed to

[1)] https://github.com/irfansha/N-Queens-data go beyond $N = 20$. The general idea is to create $N \div 2$ subproblems by setting the one queens in one of the columns in the first row. As `Somers'` algorithm uses vertical symmetry, only $N \div 2$ subproblems are created and solved in parallel using $N \div 2$ threads. The main challenge with this parallelisation is visible, as the N increases the number of threads we use increases. If the number of threads used is more than the cores available the speed is not linear, in fact it will result in overhead from thread switching.

For `Somers'` algorithm, the leaves-to-solutions ratio $q \approx 1.36 + 0.71 * N$ which is linear in input. Thus the exponential increase in the runtime of this algorithm can be inferred from the linear q. While the *FirstRow* (see Subsubsection 4.2.3.1) also chooses the fields in the same order, there is a significant difference in tree sizes produced due to the additional ALO propagation. As mentioned before, the underlying reduction in q is due to the difference in the work done inside the nodes. In other words, our AmoAlo trees are simply doing more work inside each node to produce smaller trees (to some extent). The fact that `Somers'` algorithm is roughly 15 times faster than our `GenericSAT` with *Taw* heuristics strengthens the above argumentation. Here a point to be noted is that vertical symmetry is applied to reduce the search space by half so roughly 7.5 times faster. Also, as we discussed earlier, the `Somers'` algorithm grows exponentially and the modelling of runtime (with small N values) might not be sufficient to make proper statements. Another essential point to be noted is that due to the representation of board state in three bit-field vectors for efficiency, there is loss of information regarding unsatisfiability. For example, after setting some queens on the board while the next row has open fields, some rows might be already forbidden. This lack of unsatisfiability identification results in additional processing until we find the unsatisfiability in the next row.

5.1.2 Computations for $N \geq 24$

The N-Queens problems with larger $N \geq 24$ are solved using distributed computing. It is intuitive as simple serial computation takes too long (in fact decades). However, the underlying implementation for the following approaches is based on the `Somers'` algorithm. The basic idea behind these computations is to divide the problem into subproblems by pre-placing some queens. In every implementation, at least the vertical symmetry over the chessboard is used for the reduction of search space to half. Kise, Katagiri, Honda and Yuba in [35] counted total solutions for $N = 24$ by developing and parallelising an optimised sequential algorithm using MPI. The computation for $N = 25$ is solved by INRIA, France by using Java grid computation (more details on computation are available in nqueens25, we could not find a paper on the computation).

Preußer (see Paper [47] for implementation details) solved 26-Queens and 27-Queens problems by creating many subproblems and solving them with `FPGA` devices in parallel. Two important aspects that need to be discussed in this implementation are symmetry breaking and the usage of `FPGA` devices for hardware acceleration. While only the basic symmetry over the vertical axis is considered for 26-Queens computation, other symmetries are considered for 27-Queens computation. As the symmetries over a square form the dihedral group

D_4 with 8 elements, it is possible to reduce the search space to one-eighth. The coronal pre-placements are considered (detailed explanation available in [47]) for using all the symmetries. i.e., essentially pre-placing the queens in two outer rings of the chessboard and grouping symmetric boards (only one of them is computed). The FPGA devices are a good match for the N-Queens problem considering the hardware implementations. As subproblems are small enough, using several of such specifically designed hardware devices allow massive parallelism. Design decisions for the computation and hardware details are available in the project q27.

5.1.3 Counting algorithms with non-trivial bounds

In literature, some approaches provide non-trivial upper bounds by taking advantage of inherent structure of the problem. Rivin and Zabhin [50] provide a dynamic programming approach with time and space complexity of $\mathrm{O}\left(f(n)8^n\right)$ and $\mathrm{O}\left(n^2 8^n\right)$ respectively where $f(n)$ is some lower-order polynomial. Let B, B' are two N-Boards with k placed-queens such that in both the boards same rows, columns, diagonals and anti-diagonals contain the non-attacking placed queens. Here we can observe that if B is satisfiable then B' is also satisfiable. A solution to B' can be easily completed by the placing the same additional queens placed on the completed B. The dynamic programming approach uses this observation and groups partial solutions into equivalence classes based on the lines (rows, columns, diagonals, and anti-diagonals) that are placed. While the upper bound for time complexity is simply exponential, it uses exponential space resulting in difficulty of implementation for larger N.

Engelhardt [20] further discuses the improvements over Rivin-Zabhin's algorithms by considering look-ahead and exploiting symmetries. An idea was presented to look at N-Queens solutions as the solutions on a modular N-Queens problem (first considered in [45]) with some conflicts i.e. a diagonal or antidiagonal (on the underlying torus) containing two placed queens. A group based search for computing all the solutions was also presented however the pratical improvement of implementation for larger N is still open. Pratt [46] provides an implementation based on permanent computation and formulating N-Queens as an obstruction matrix. The approach proposed provides an implementation with exponential time complexity $\mathrm{O}\left(n^2 32^n\right)$ and quadratic space complexity.

5.2 #SAT

In this section, we extend our discussion from Section 2.2 on counting with connected components and other techniques used by modern DPLL algorithms over DLL solvers for better performance. We consider only sharpSAT, a DPLL based solver as it inherits advances from different solvers and is considered as state of the art. We discussed earlier that loosely constrained problem with higher solution count need not be difficult to count.

5.2.1 Counting with connected components

In this section, we briefly discuss counting with connected components.

Definition 5.2.1 *A* ***constraint graph*** *is a pair $G = (V, E)$ of a* CNF *formula F, where the set of* vertices *V is the set of* variables *of F, and E is the set of* edges *connecting two vertices if the corresponding variables appear together in some clause of F.*

Definition 5.2.2 *A pair $C = (V_c, E_c)$ is a* ***component*** *of constraint graph $G = (V, E)$, if $V_c \subseteq V$ and $E_c \subseteq E$.*

Definition 5.2.3 *Components $C_i = (V_i, E_i)$ and $C_j = (V_j, E_j)$ are* ***disjoint components*** *if there is no edge (a, b) such that $a \in V_i$ and $b \in V_j$.*

Definition 5.2.4 *A* ***connected component*** *$CC = (V_{cc}, E_{cc})$ is a component of the constraint graph $G = (V, E)$, if CC is a maximal subgraph of G and every pair of vertices $(a, b) \in V_{cc}$ has a path from a to b.*

Let G be the `constraint graph` of a CNF formula F, partitioned into `disjoint components` $G_1, G_2 \dots G_k$. Let $F_1, F_2 \dots F_k$ be sub-formulas of F, which are corresponding formulas of the `disjoint components` $G_1, G_2 \dots G_k$. The complete satisfying assignments of F can be computed by the product of total satisfying assignments of k sub-formulas $F_1, F_2 \dots F_k$ due to their disjointness. `relsat` (presented in [32]), first implemented this approach for exact model counting, moreover connected components are detected to exploit its advantage fully. Further, the connected components are detected dynamically as the underlying `DPLL` procedure extends a partial assignment. Every subproblem is checked for unsatisfiability, as F is unsatisfiable if any one of the sub-formulas is unsatisfiable (due to the disjointness of sub-formulas). Further improvements are made by dynamically jumping from one sub-problem to another if the current one becomes less constrained. As part of `DPLL` counting, if a partial assignment is found to be satisfied, then a solution count of 2^{n-t} is returned where n is the total number of variables of F, and t is the number of variables assigned.

5.2.2 sharpSAT

`sharpSAT` is developed by Marc Thurley [58]. It is based on `DPLL` algorithm for model counting (presented in [8]) combined with component decomposition in `relsat` [32] and component caching with branching heuristics in `cachet` [53, 54]. It outperforms `relsat` and `cachet` by improving upon inherited techniques component caching and implicit BCP (boolean constraint propagation), and is also considered as the state-of-the-art for model counting.

5.2.2.1 Component caching

For component caching, `sharpSAT` uses the packing scheme `Hybrid coding omitting binary clauses packing scheme` (HCOP) instead of the default

`Standard scheme` (STD) as in `cachet`. Here we provide a brief explanation of the coding, more detailed explanation is provided in [58].

Definition 5.2.5 *A* `component` *$SC = (V_{sc}, E_{sc})$ is a* ***sound component****, if the corresponding* CNF *formula F has atleast two unassigned literals.*

In HCOP, only `sound components` are cached, as zero length clauses denote conflicts and BCP handles unit clauses. HCOP uses `Hybrid coding scheme` (HC), `Omitting binary clauses`, and `Packing scheme` to encode the clauses efficiently for caching. Let C be a `component` and F be its corresponding CNF formula with $\text{var}(F)$ and $\text{cls}(F)$ be the set of variables and the set of clauses respectively. Let $\text{var}_{id}(F)$ and $\text{cls}_{id}(F)$ be the set of indices for $\text{var}(F)$ and $\text{cls}(F)$. In HC, F is reduced to a tuple of strings (a, b) where a and b corresponds to the indices of $\text{var}(F)$ and $\text{cls}(F)$ respectively. For example, $F = (x_1 \vee x_2 \vee x_3) \wedge (-x_2 \vee x_4)$, then $\text{var}_{id}(F) = \{1, 2, 3, 4\}$ and $\text{cls}_{id}(F) = \{1, 2\}$. Further F is reduced to tuple (a, b), where $a = (1, 2, 3, 4)$ and $b = (1, 2)$. The size of codes cached are further reduced by omitting the storage of binary clauses of the original formula as they can be reconstructed easily from the set of variables of the component referred to as `Omitting binary clauses`. Further, `Packing scheme` is used to pack the tuple (a, b) into $(\hat{a}, \hat{b})$ by bitshifting a and b respectively such that the identifiers contain information in only n least significant bits. Every component is encoded using HCOP, reducing the required cache memory. Further, an absolute bound on cache size is set to handle excessive cache usage for hard formulas. `sharpSAT` also improves on Variable State Independent Decaying Sum (VSIDS) heuristics (see Paper [54] for a detailed discussion on the implementation of the heuristics) to stabilise the cache size without any significant overhead.

5.2.2.2 Implicit boolean clause propagation

Unit Propagation (UP) heuristics (see Paper [40] for the impact of unit propagation on SAT problems) generally used in SAT and #SAT solvers, which is branching on a variable x maximizing boolean clause propagation (BCP) by its assignment (*look ahead* in a sense). `sharpSAT` improves upon the UP heuristics by finding all the failed literals independent of the branching variable and conflict clauses are learned directly from them. For example, the failed literal test is applied to a set of literals by setting each literal x to $\top$ and $\bot$. If setting x to $\top$ results in unsatifiability then assert $x = \bot$. Finding all such forced literals reduces both the resulting component analysis and caching significantly. Additionally, only literals reduced to binary in the most recent BCP call are considered. Thurley [58] analyses the impact of implicit BCP in detail.

5.2.3 Experimental results and heuristic argumentation

The `sharpSAT` solver provides three (main) options `-noPP`, `-noCC`, and `-noIBCP` for turning off the techinques that are used in the processing. Here the command `-noPP` represents turning off the preprocessing option. The command `-noCC` represents turning off the component caching option. Whereas the command

PP	**CC**	**IBCP**	**q**	**qt**
N	N	N	≈ 2.78	$\approx (1.43e - 05 \cdot N)$
Y	N	N	≈ 2.78	$\approx (1.36e - 05 \cdot N)$
N	Y	N	≈ 2.777	$\approx (1.25e - 05 \cdot N)$
N	N	Y	≈ 1.44	$\approx (5.36e - 05 \cdot N)$
Y	Y	N	≈ 2.77	$\approx (1.32e - 05 \cdot N)$
Y	N	Y	≈ 1.438	$\approx (5.4e - 05 \cdot N)$
N	Y	Y	≈ 1.44	$\approx (5.43e - 05 \cdot N)$
Y	Y	Y	≈ 1.44	$\approx (5.32e - 05 \cdot N)$

Figure 5.1: `sharpSAT` analysis for the counting N-Queens problem with all combinations of options.

`-noIBCP` represents turning off Implicit boolean clause propagation (implicit BCP). The preprocessing is a common technique used in SAT solvers and we discussed the additional techniques component caching and implicit BCP briefly in previous sections. We generated data for all the 8 combinations of the 3 options preprocessing, component caching and implicit BCP to investigate their impact on the counting N-Queens problem. The data is available in N-Queens-data[2)] and the instructions for data generation are provided in each file for reproduction. Similar to `GenericSAT` data, we model the leaves-to-solutions q ratio and runtime-to-solutions ratio qt. For each combination of the 3 options, we generated the data for the N-Queens problem up to $N = 16$ (already takes a couple of hours). A point to noted is that the `sharpSAT` give 2 solutions for 1-Queens problem when used with options other than preprocessing. This seems to be a bug (only for $N = 1$), rest of the data seems to be intact.

Table 5.1 provides modeling of leaves-to-solutions ratio (q) and time-to-solutions ratio (qt) for N-Queens problems. The `sharpSAT` provide the number of conflicts as part of data, which is essentially the number of unsatisfiable leaves, so combined with total solutions, we get the total number of leaves. Looking at the leaves-to-solutions ratio, it is clear that for every combination of the option the q is some constant factor. The component caching and preprocessing does not seem to have much impact on the sizes of the trees. However, the implicit BCP seems to reduce the q significantly performing much better than *Taw* heuristics in `GenericSAT`, which produces the smaller trees. The Figures 5.2 and 5.3 show the impact of IBCP on the leaves-solutions ratio. Intuitively this should be the case, as implicit BCP by its nature identifies the forced literals at every branching decision, thus increasing propagation. However, this also creates an overhead as at every branching decision some set of literals are tested if they are forced literals. It indeed seems to create a significant overhead, which is evident by looking at the qt that increases significantly when implicit BCP is used (can be observed in the Table 5.1). While the leaves-to-solutions ratio seems to be better than the AmoAlo trees with the *Taw* heuristics, the runtime-to-solutions ratio seems to be worse. The longer runtime seems to be due to the significant overhead caused by additional

[2)] https://github.com/irfansha/N-Queens-data

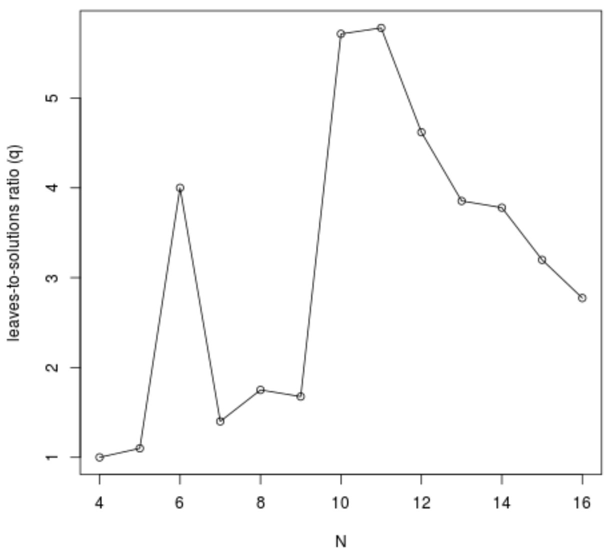

Figure 5.2: Leaves-to-solutions ratio for `sharpSAT` with only preprocessing and component caching.

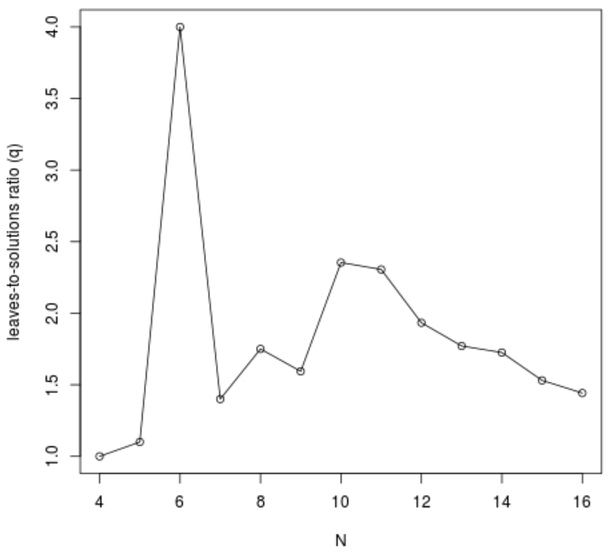

Figure 5.3: Leaves-to-solutions ratio for `sharpSAT` with all options.

techniques, which strengthens our Conjecture (2.4.2) that `DLL` algorithms are equally good to other approaches for the N-Queens problem. We can also argue that (as we did for `GenericSAT` data), the qt while looking linear it might not be the case when the counting N-Queens problem is large enough. By looking at the complexity analysis of the `sharpSAT` implementation, we can perhaps, properly state the comparison between these two approaches. On the other hand, from this data, it is empirically evident that the trees produced are in a sense (i.e., by considering conflicts) output-polynomial in size and the algorithms have output-polynomial runtime.

5.3 #CSP

Any combinatorial problem can be formulated as a #CSP problem to count the total satisfying solutions. In fact, #SAT, counting satisfying assignments for a propositional formula, is a restricted subclass of #CSP where the values of variables are restricted to boolean values. The trees that are produced by the #CSP solvers are usually n-ary trees, i.e., n branches at each node. The binary branching in QBT is more flexible, and it is possible to simulate the n-ary branching by choosing the appropriate branching variables. For the N-Queens problem, we consider a state-of-the-art #CSP solver, `Minion`, and investigate the sizes of the trees produced. Similar to our framework, there are several branching heuristics available in `Minion`. So we investigate the change in tree growth with the change in branching heuristics and compare with our AmoAlo trees.

5.3.1 Minion

`Minion` [22] is a scalable general-purpose constraint solver with a focus on matrix models as the input language for optimised implementations. The input language for matrix models provides some fundamental constraints over integer domains. It differs from general constraint toolkits (e.g. `Gecode`) by providing efficient (kind of *blackbox*) implementation with `models` as input instead of programs. This modelling allows for scalable and fine tuned implementation at the cost of relatively less flexibility and functionality. It attempts to achieve the *model and run* paradigm similar to propositional satisfiability (SAT) and mathematical programming (MP). Gent [22] shows that `Minion` is 1 to 2 orders faster than the state-of-the-art constraint toolkits. The experiments discussed in the paper are more specific to constraint solving rather than counting all solutions. Further advances are made in `Minion` such as adding watched literals for propagation (see Paper [23] for further explanation) and improved data structures for generalised arc consistency (GAC)(see Papers [24, 43] for explanations). In #CSP, the instances are simplified recursively by splitting into subproblems and applying propagation on each instance. The choice of branching variable for splitting can be done either statically or with dynamic heuristics, which is close to our implementation. We briefly discuss the branching heuristics used in `Minion` and model tree growth from the different heuristics in the following sections.

5.3.1.1 Branching heuristics

`Minion` provides the following branching heuristics:

1. `static` (default) branching chooses the variables lexicographically.
2. `sdf` branching chooses variable with smallest domain.
3. `ldf` branching chooses variable with largest domain.
4. `wdeg` branching is a weight based degree variable heuristic.
5. `domoverwdeg` branching is also a weight based degree variable heurisic but a refinement over `wdeg`.

Further options are available for the static ordering heuristics based on how the ties between variable are handled. For example, if two variables have the same `measure` then either the lexicographically smaller variable is chosen, or one of them is chosen in random. The `smallest domain first` (sdf) heuristic breaks the ties lexicographically whereas `sdf-random` breaks randomly. However, we do not consider the additional heuristics that arise from random tie-breaking. The main focus here is to look at the smallest and largest trees, but the random breaking does not produce either of them.

Both static heuristics and dynamic heurstics are considered, of them, `wdeg` and `domoverwdeg` are weighted degree variable heuristics. We model and empirically analyse the tree growth for each of these heuristics.

5.3.2 Experimental data and heuristic argumentation

The `Minion` provides the branching heuristics discussed above. As it takes models as input, we use the `nqueens-JFP` generator for generating models of the N-Queens problems. The generator `nqueens-JFP` is provided along with the software. For each heuristic, we generated data for the N-Queens problem up to $N = 17$ (as this already takes a couple of hours). However, for the `ldf` branching heuristic, we only generated the data upto $N = 15$ (as $N = 16$ is already taking too long). The data for each of the heuristics are provided in N-Queens-data[3)]. The instructions for the generation of data are provided in the files for reproduction. To compare the trees produced in `Minion` to AmoAlo trees, we consider nodes-to-solutions ratio instead of the leaves-to-solutions ratio. The statistics produced by the `Minion` does not include the number of leaves, so we directly consider nodes for comparison. A point to be noted is that the notion we use to investigate the complexity of AmoAlo trees, the leaves-to-solution ratio is essentially half of the nodes-to-solutions ratio (thus allowing for comparison).

The Table 5.4, presents the modelling of the nodes-to-solutions ratio qn and runtime-to-solutions ratio qt for each branching heuristics. From the modelling, it is clear that it can be observed that qn(N) is linear in N for all the heuristics except `ldf` which seems to be polynomial. It also seems that the weight-based

[3)]https://github.com/irfansha/N-Queens-data

Heuristics	**qn**	**qt**
default	$\approx 1.21 * N$	$\approx 5.26e-06 * N$
sdf	$\approx 0.87 * N$	$\approx 4.83e-06 * N$
wdeg	$\approx 2.20 * N$	$\approx 9.67e-06 * N$
domoverwdeg	$\approx 0.99 * N$	$\approx 5.26e-06 * N$
ldf	$\mathrm{O}(N^{5.68})$	$\mathrm{O}(N^{5.27})$

Figure 5.4: `Minion` data with different branching heuristics.

degree heuristics do not improve the performance by much. This lack of improvement seems to be different from the effect of the dynamic heuristics such as the *Taw* heuristics in the `GenericSAT`, where we observed significant improvement. By comparing the heuristics in `GenericSAT` and `Minion`, it can be argued that choosing variables considering future propagation (for example the *Taw* heuristics) performs better. Considering heuristics like *Taw* heuristics, which consider future propagation might improve the performance of constraint solvers. Despite the branching heuristics having qn = $\mathrm{O}(N^{5.68})$, which is worse than the *Antitaw* heuristics in the AmoAlo trees, all the trees here are output-polynomial in size. The output-polynomial tree sizes strengthen that the minion also has an output-polynomial runtime (which is clear from the q).

5.4 Counting with knowledge compilers

Knowledge compilation is an attempt to deal with computational intractability of general propositional reasoning (detailed discussion in [14, 16]). Here a propositional theory is compiled into a target language which can be used to answer a large number of queries in polytime that are otherwise intractable in the original theory. The computational effort in knowledge compilation is divided into two phases, offline and online. The propositional theory is compiled into a target tractable language in the offline phase, and then in the online phase, exponential non-trivial queries can be asked. Darwiche [16] provides a knowledge compilation map by analysing a large number of existing target languages considering various properties such as succinctness, polytime transformations, and queries. Selman [55] proposed Horn theory as the target compilation language; however, not every propositional theory can be compiled into a Horn theory. Thus the propositional language is approximated (in terms of bounds) by compiling into two different Horn theories with lower and higher logical strength. Darwiche [13] instead proposed decomposable negation normal form (DNNF) as the target language with the advantage that every propositional theory can be compiled into it. An algorithm is proposed to compile any propositional theory into DNNF and also showed that a DNNF could be compiled from a clausal form with bounded treewidth in linear time. DNNF allows polytime tests for consistency and clausal entailment, and polytime algorithm for model enumeration. Approximations for DNNF compilations are provided when the resulting DNNF is too large.

Later in this section, we provide definitions for negation normal form (NNF) and its subsets briefly. Deterministic DNNF (d-DNNF) (introduced in [14]), however, is of our primary focus as it is the most succinct (or compact) target language among the subsets of NNF that allow polytime counting.

5.4.1 d-DNNF

The CNF and DNF are the subsets of flat NNF where the flatness is due to the restriction of height. Here we consider nested NNF where height is not a restriction, but other restrictions such as determinism, smoothness, and decomposition are imposed to obtain the sub-classes DNNF and d-DNNF. We only provide definitions specific to counting. Darwiche [16] provided detailed explanations and proofs on various subsets of NNF.

Definition 5.4.1 *Let S be a set of propositional variables. A sentence S is in **NNF** if it is a directed acyclic graph (DAG) with inner nodes labled with $\wedge, \vee$ and leaves labled with true, $false$, X or $-X$ where $X \in S$.*

In an NNF, a node labeled with $\wedge$ representing a conjunction is an **and-node**, and a node labeled with $\vee$ representing a disjunction is a **or-node**. $\mathrm{Var}(C)$ is the set of **variables** of the descendants of C and $\xi(C)$ represents the set of **children** of C. We now define some of the restrictions that are imposed upon NNF to obtain sub-classes.

Definition 5.4.2 *Let C be an and-node in a* NNF *and $C_i, C_j \in \xi(C)$. The* NNF *is said to be **decomposable** if for each C in* NNF *and $i \neq j$, $\mathrm{Var}(C_i) \cap \mathrm{Var}(C_j) = \emptyset$.*

In other words, for each and-node in NNF no two children of C share common variables.

Definition 5.4.3 *Let C be an or-node in an* NNF *and $C_i, C_j \in \xi(C)$. The* NNF *is said to be **deterministic** if for each C in* NNF *and $i \neq j$, $C_i \wedge C_j = false$.*

In other words, for each or-node in NNF every two children of C are logically contradictory.

Definition 5.4.4 *Let C be an or-node in a* NNF *and $C_i, C_j \in \xi(C)$. The* NNF *is said to be **smooth** if for each C in* NNF *and $i \neq j$, $\mathrm{Var}(C_i) = \mathrm{Var}(C_j)$.*

In other words, for each or-node in NNF, every child of C has the same variables. Using these definitions, we now formally define some sub-classes of the NNF.

Definition 5.4.5 *An* NNF *satisfying decomposability is a* **DNNF***.*

Definition 5.4.6 *A* DNNF *satisfying determinism is a* **d-DNNF***.*

Definition 5.4.7 *A* d-DNNF *satisfying smoothness is a **smooth*** **d-DNNF***.*

d-DNNF as defined above is a subset of NNF satisfying decomposition and determinism. It is the most succinct (proved in [16]) class, which allow polytime counting, i.e., the language d-DNNF has the shortest representation among the other subsets that allow polytime counting. Darwiche [14] proves that d-DNNF are more compact than OBDDs and provides an algorithm to convert from OBDDs to d-DNNF. They also provide lower bounds on the size of d-DNNF given bounds on tree representation. Further, the counting is not done directly on the d-DNNF tree, but a corresponding counting graph is constructed. In general smooth d-DNNF are considered for counting for simplicity. The main idea behind constructing counting graph is to use the underlying backtracking structure to count all the solutions. The counting graph is constructed by changing the labels of the d-DNNF tree.

Definition 5.4.8 *A* ***counting graph*** *G for a smooth* d-DNNF *is constructed by labeling*

- *every and-node in* d-DNNF *with* $*$
- *every or-node in* d-DNNF *with* $+$
- *every satisfied leaf with* 1
- *every unsatisfied leaf with* 0*.*

The main idea behind counting with d-DNNF is converting a propositional theory p (usually CNF) into a smooth d-DNNF and constructing corresponding counting graph G. By traversing through the counting graph, partial derivatives are computed for each variable (see Paper [51] for the complete algorithm). This basic idea is used in the following knowledge compilers as well. Darwiche [13] provided an algorithm to convert CNF into d-DNNF, which is further improved in [15] and implemented in `C2D` compiler. `C2D` is considered as standard for compiling into d-DNNF with comparable representation size. On the other hand, `DSHARP` exploits advances in #SAT based on `sharpSAT` performing better than `C2D` (comparision available in [42]). In line with the literature, `DSHARP` seems to perform relatively well compared to `C2D`. So we consider only `DSHARP` and model the size of d-DNNF representation for N-Queens problem.

5.4.2 DSHARP

`DSHARP` [42] is based on the `sharpSAT` (see Subsection 5.2.2) and is a CNF to DNNF compiler. It takes advantage of the advances in `sharpSAT` and produces a better result compared to the state-of-the-art `C2D`. As discussed above, CNF is compiled into d-DNNF and solutions are counted using the counting graph. The techniques for improvements are the same as in the `sharpSAT`, i.e., component caching, preprocessing and implicit BCP (see Subsection 5.2.2).

The d-DNNF trees produced for N-Queens problems are of our main concern. As these trees do not have unsatisfiable nodes, the trees are more compact than our AmoAlo trees. A point to be noted is that the assignments of the variables are also represented in the graph resulting in larger trees. However,

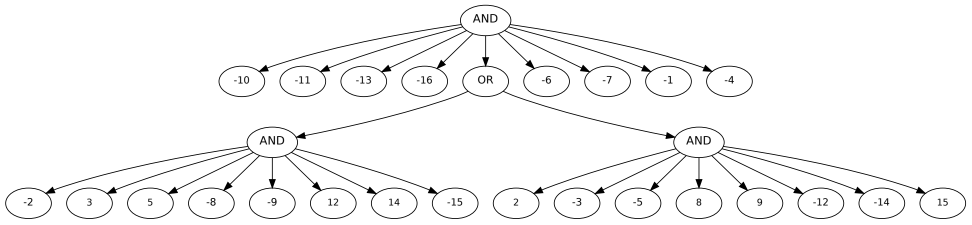

Figure 5.5: d-DNNF graph for 4-Queens problem.

the critical question here is whether the sizes of the trees are asymptotically less than the number of solutions. The smaller tree sizes provide a possibility to go beyond enumeration and into counting.

5.4.2.1 Empirical analysis and discussion

The DSHARP solver provides similar options like the sharpSAT such as turning off preprocessing, component caching and Implicit BCP. However, here the d-DNNF trees that are produced for the N-Queens problem and their sizes are of primary concern. Usage of the options does not seem to affect the sizes of the d-DNNF trees much so we only consider the default options, i.e. allowing all the additional options and produce the trees. The node count of d-DNNF trees for the N-Queens problem are computed upto $N = 14$. As the generation of these d-DNNF trees require much main memory, for $N = 14$ it already requires around 50GB, so we did not compute for $N \geq 15$. The data is available in N-Queens-data[4] and the instructions for generation are provided in the data file for reproduction.

From the data produced by DSHARP, the nodes-to-solutions ratio qn is a constant factor and approximately equal to 2.7. Note that this is nodes-to-solutions ratio where we generally consider leaves-to-solutions ratio q. So these trees are already smaller than AmoAlo trees and the trees produced by sharpSAT. As mentioned earlier, the trees produced by DSHARP also include information on assignments.

For example, Figure 5.5 is a d-DNNF graph for $N = 4$, from which it is clear that many of the nodes are for representing the assignment information. So, this indicates the possibility that the qn ratio can be even smaller providing a way to investigate if it is possible to go beyond the simple enumeration for the N-Queens problem. In Figure 5.6, we can observe that the nodes-to-solutions ratio is decreasing with increase in N strengthening our argument. In Section 2.4, we discussed that we are considering enumeration instead of counting as we are not aware of any counting algorithms and the relative hardness of counting and enumeration. However, these trees, if they have the total number of nodes asymptotically smaller than the solutions count then counting is possible. The primary criteria for the existence of counting algorithms for the N-Queens problem (or any counting problem in general) are that there exist some tree representations that are asymptotically smaller than the solutions count. The

[4] https://github.com/irfansha/N-Queens-data

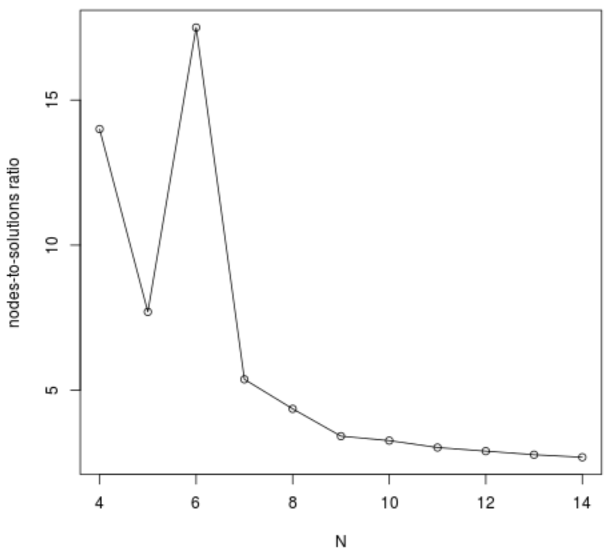

Figure 5.6: Leaves-to-solutions ratio `DSHARP`.

future work here is to reduce these trees into AmoAlo trees or vice-versa to investigate if some representations allow counting of solutions instead of enumeration.

Chapter 6

Ongoing and future work

In previous chapters, we provided a framework to investigate the complexity of the counting N-Queens problem and discussed the `GenericSAT` framework for experimentally investigating the conjectures on the sizes of AmoAlo trees. While we provided heuristic argumentation about output-polynomiality of the problem, we did not prove it.

In this chapter, we discuss our ongoing research for making a stronger hypothesis and possibly answering the conjectures. Further, we summarise our contribution to the N-Queens problem and discuss future work.

6.1 Ongoing work

In Section 4.3, we observed that exponential unsatisfiable subtrees occur in AmoAlo trees generated by `GenericSAT` resulting in lack of polynomial delay. First possible step to reduce sizes of AmoAlo trees is by avoiding the exponential unsatisfiable subtrees. It would be interesting to see where the unsatisfiability occurs and to investigate ways to avoid such cases. As the completion problem is NP-Complete, it might not be possible to avoid all of them. However, it is possible to identify at least some of those unsatisfiable subtrees with some polynomial-time unsatisfiability tests. We use `Tulip` to visualise the AmoAlo trees and look at the unsatisfiability. For example, the Figures 6.1,6.2 and 6.3 are the AmoAlo trees for the 8-Queens problem visualised by `Tulip`.

The observations are interesting as the largest unsatisfiable subtrees for *SquareEnum* and *Antitaw* heuristics seems to occur in the leftmost unsatisfiable subtree (see Figures 6.2 and 6.3), whereas it is non-specific for the *Taw* heuristics (see Figure 6.1). Here the nodes with black colour are unsatisfiable nodes and with yellow colour are satisfiable nodes. From further investigations, these largest unsatisfiable subtrees for *SquareEnum* and *Antitaw* heuristics are Rooks unsatisfiable cases. Given a partially filled N-board with k queens, we cannot place $N - k$ Rooks on it such that no piece attack each other. The fundamental invariant of the Rooks problem is that no two rooks can be placed in the same row or column. This observation is essential because the satisfiability of Rooks problem can be formulated as the existence of perfect matching for a corresponding bipartite graph. Efficient polynomial-time tests exist for

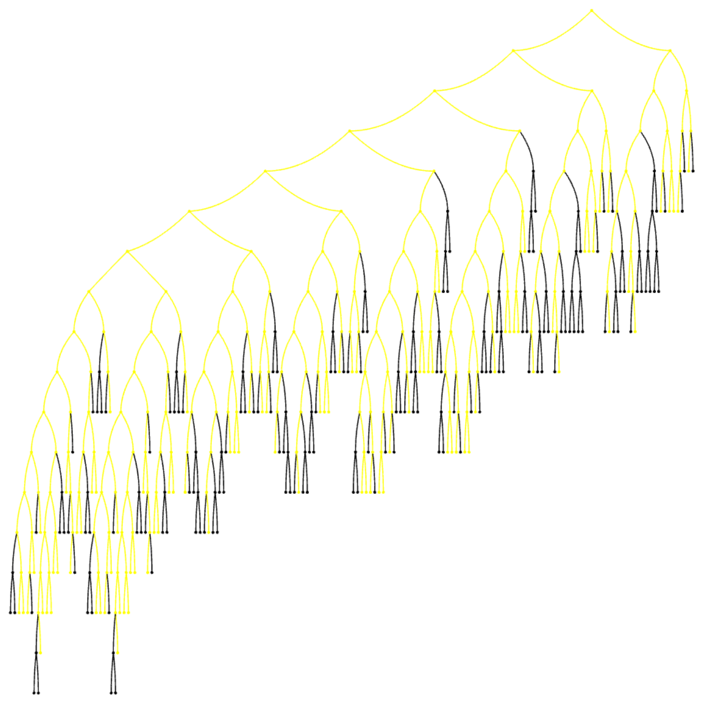

Figure 6.1: 8-Queens AmoAlo tree with *Taw* heuristics.

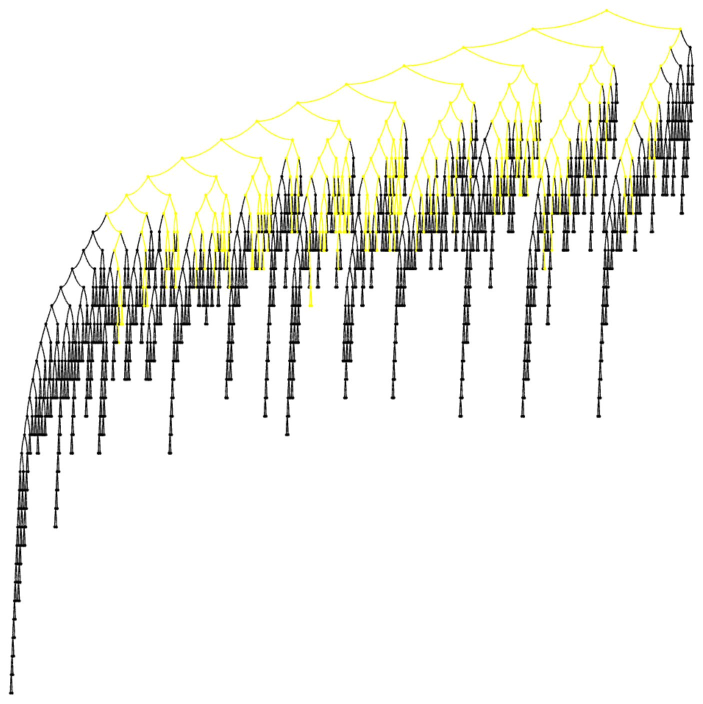

Figure 6.2: 8-Queens AmoAlo tree with *SquareEnum* heuristics.

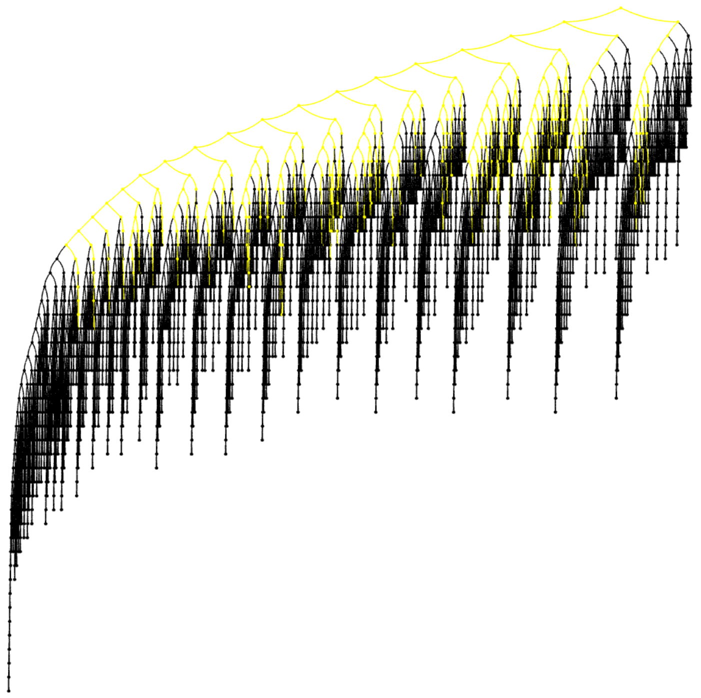

Figure 6.3: 8-Queens AmoAlo tree with *Antitaw* heuristics.

deciding the existence of a perfect matching. Thus these unsatisfiable subtrees can be avoided using polynomial tests. The Queens constraints are a combination of the Rooks problem and Bishops problem. The fundamental invariant of Bishops problem is that no two bishops can be placed in the same diagonal (and anti-diagonal). A natural question to ask would be *Does N-Bishops unsatisfiable cases exist too?*, it seems to be the case. A resulting board, with some placed and forbidden fields, can be Bishops unsatisfiable. Given a partially filled N-board with k queens, if we cannot place $(N-k)$ bishops on the board such that no piece attack each other is a *bishops unsatisfiable case*. This unsatisfiability can be avoided by formulating the board as a bipartite graph (but now considering the diagonal constraints) and finding if there exists a perfect matching.

To investigate the impact on the reduction of tree sizes by these tests, a similar but less powerful unsatisfiability test is implemented. In a board, if there are not enough open diagonals or anti-diagonals to placed rest of the queens, then it is already bishops unsatisfiable. This simple test can be performed by looking at the ranks of the diagonals and anti-diagonals, which can be implemented in linear time. The data with *Taw* heuristics and additional *not enough diagonals* test is available in `oklibrary`. The leaves-to-solutions ratio $\mathrm{q}(N) = 2.315$ and time-to-solutions ratio is $\mathrm{qt}(N) = 4.45e-06$. While there is no significant change in q, there is some reduction in tree size. Remember that we have exponential solutions, so any (small) changes in q represent larger changes in the tree sizes. Another important observation is that the runtime is slightly less with the unsatisfiability test, which seems to be due to the avoided unsatisfiable subtrees. We conjecture that implementing Rooks and Bishops unsatisfiable tests will reduce the tree sizes significantly without much overhead.

We also want to look at the impact of these unsatisfiability tests on the other branching heuristics. It is possible that there can be a significant reduction in tree sizes for the other branching heuristics performing better than the *Taw* heuristics. In Figure 6.2, it can be observed that the *SquareEnum* heuristics has many unsatisfiable subtrees. Earlier, we observed that many of these subtrees are Rooks unsatisfiable cases, thus indicating a possibility of significant size reduction. Apart from unsatisfiability, to gain more insight into the propagation, we are improving our framework to collect statistics from different phases of the propagation inside each node. We conjecture that understanding the propagation better will help to estimate bounds on the sizes of the AmoAlo trees.

6.2 Our contribution

To investigate the complexity of the counting N-Queens problem, first, a mathematical modelling of the N-Queens problem is provided. Then, a class of trees called QBT trees and its various subclasses with unsatisfiability and propagation as additional parameters are introduced. Some basic lemmas for QBT and subclasses of QBT trees provided. Further, AmoAlo trees, a subclass of trees, i.e., the trees with both unsatisfiability and propagation that arise from AMO and ALO constraints of the N-Queens problem are discussed in detail. `GenericSAT`, a SAT based framework to experimentally analyse the conjectures on AmoAlo trees and efficiently count all the solutions of the N-Queens problem is introduced and discussed. For the investigation of sizes of AmoAlo trees, 3 heuristics specific to the counting N-Queens problem are developed. Their implementations, along with the existing *Taw* heuristics, are provided and discussed. The AmoAlo trees with different branching heuristics are heuristically shown to be output-polynomial in size, and the algorithms are output-polynomial in time. Among the branching heuristics, it is shown that the *Taw* heuristics generate smaller trees with leaves-to-solutions ratio $q = 2.3$ suggesting the existence of output-polynomial time algorithms. Further, the trees produced by state-of-the-art solvers are discussed and compared to AmoAlo trees. We also empirically observed that the trees produced by these state-of-the-art solvers are also output-polynomial in size. Finally, possible approaches to avoid unsatisfiability and go beyond enumeration paradigm are discussed.

6.3 Future work

One of the main research problems for the N-Queens problem is to develop an efficient counting algorithm that goes beyond simple enumeration paradigm.

However, the existence of efficient counting algorithms is still unclear. The existence of such algorithms can be argued that, if there exist some algorithms that generate tree sizes asymptotically smaller than the total number of solutions, then counting is possible. By looking at the nodes-to-solutions ratio in DSHARP data, this seems to be a possibility. Remember that the d-DNNF trees generated here also specify the assignments thus resulting in larger trees. In other words, there can be some underlying tree representation that is asymptotically smaller than solutions count and investigating this possibility is one aspect of our future work.

sharpSAT seems to perform well (i.e. counting instead of enumerating) on the N-Rooks problem, a related chessboard problem. Using Cube-and-Conquer, a hybrid approach, we can take advantage of this efficiency of sharpSAT solver. The Cube-and-Conquer brings together the strengths of two classes of SAT solvers, look-ahead and CDCL solvers. The look-ahead solvers are good in partitioning the problem into equally *hard* subproblems whereas CDCL solvers are good in solving specific (local) instances. The Rooks constraints are subset of Queens constraints (discussed in Section 6.1) over the fields of board. So it is possible to create subproblems such that they are close to the N-Rooks instances to use the efficiency of solvers such as sharpSAT. Our design for integrating the Cube-and-Conquer approach into GenericSAT is to use branching heuristics such that the created subproblems are (almost) Rooks boards, i.e., we try to remove the diagonal constraints as much as possible. Then use the efficient CDCL solvers like sharpSAT on the (Rook-like) subproblems. We conjecture that the (underlying) trees generated in this approach will be smaller (perhaps asymptotically smaller than solutions), resulting in an efficient implementation.

Bibliography

[1] Tanbir Ahmed, Oliver Kullmann, and Hunter Snevily. On the van der Waerden numbers $\mathrm{w}(2; 3, t)$. *Discrete Applied Mathematics*, 174:27–51, September 2014. `doi:10.1016/j.dam.2014.05.007`.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. URL: `http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264`.

[3] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004. URL: `https://doi.org/10.1613/jair.1410`, `doi:10.1613/jair.1410`.

[4] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009. URL: `https://doi.org/10.1016/j.disc.2007.12.043`, `doi:10.1016/j.disc.2007.12.043`.

[5] Max Bezzel. Proposal of eight queens problem. *Berliner Schachzeitung*, 3:363, 1848.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[7] Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[8] Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res.*, 10:457–477, 1999. URL: `https://doi.org/10.1613/jair.601`, `doi:10.1613/jair.601`.

[9] Josh Buresh-Oppenheim and Toniann Pitassi. The complexity of resolution refinements. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, page 138, 2003. URL: `https://doi.org/10.1109/LICS.2003.1210053`, `doi:10.1109/LICS.2003.1210053`.

[10] Robert Burton and Jeffrey E. Steif. Non-uniqueness of measures of maximal entropy for subshifts of finite type. *Ergodic Theory and Dynamical Systems*, 14(2):213235, 1994. doi:10.1017/S0143385700007859.

[11] Paul J. Campbell. Gauss and the eight queens problem, a study in miniature of the propagation of historical error. *Historia Mathematica*, 4:397–404, 1977. doi:10.1016/0315-0860(77)90076-3.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971. URL: https://doi.org/10.1145/800157.805047, doi:10.1145/800157.805047.

[13] Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 284–289, 1999. URL: http://ijcai.org/Proceedings/99-1/Papers/042.pdf.

[14] Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001. URL: https://doi.org/10.3166/jancl.11.11-34, doi:10.3166/jancl.11.11-34.

[15] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332, 2004.

[16] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. URL: https://doi.org/10.1613/jair.989, doi:10.1613/jair.989.

[17] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. URL: https://doi.org/10.1145/368273.368557, doi:10.1145/368273.368557.

[18] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. URL: http://doi.acm.org/10.1145/321033.321034, doi:10.1145/321033.321034.

[19] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987. URL: https://doi.org/10.1016/0004-3702(87)90002-6, doi:10.1016/0004-3702(87)90002-6.

[20] Matthias R. Engelhardt. A group-based search for solutions of the n-queens problem. *Discrete Mathematics*, 307(21):2535–2551, 2007. URL: https:

//doi.org/10.1016/j.disc.2007.01.007, doi:10.1016/j.disc.2007.01.007.

[21] R. Garnier and J. Taylor. *Discrete Mathematics: Proofs, Structures and Applications, Third Edition*. CRC Press, 2009. URL: https://books.google.co.uk/books?id=gy7NBQAAQBAJ.

[22] Ian P. Gent, Chris Jefferson, and Ian Miguel. MINION: A fast, scalable, constraint solver. In *ECAI 2006*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102, Amsterdam, The Netherlands, 2006. IOS Press.

[23] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, pages 182–197, 2006. URL: https://doi.org/10.1007/11889205_15, doi:10.1007/11889205_15.

[24] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 191–197, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-029.php.

[25] Ian P. Gent, Christopher Jefferson, and Peter Nightingale. Complexity of n-queens completion. *J. Artif. Intell. Res.*, 59:815–848, 2017. URL: https://doi.org/10.1613/jair.5512, doi:10.1613/jair.5512.

[26] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2293–2299, 2007. URL: http://ijcai.org/Proceedings/07/Papers/369.pdf.

[27] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 54–61, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-009.php.

[28] Carla P. Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 274–289, 2004. URL: https://doi.org/10.1007/978-3-540-30201-8_22, doi:10.1007/978-3-540-30201-8_22.

[29] Michael C. Horsch and William S. Havens. Probabilistic arc consistency: A connection between constraint reasoning and probabilistic reasoning. In *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000*, pages 282–290, 2000. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=33&proceeding_id=16`.

[30] Kazuo Iwama. CNF satisfiability test by counting and polynomial average time. *SIAM J. Comput.*, 18(2):385–391, 1989. URL: `https://doi.org/10.1137/0218026`, `doi:10.1137/0218026`.

[31] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986. URL: `https://doi.org/10.1016/0304-3975(86)90174-X`, `doi:10.1016/0304-3975(86)90174-X`.

[32] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 157–162, 2000. URL: `http://www.aaai.org/Library/AAAI/2000/aaai00-024.php`.

[33] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA.*, pages 203–208, 1997. URL: `http://www.aaai.org/Library/AAAI/1997/aaai97-032.php`.

[34] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 317–331, 2004. URL: `https://doi.org/10.1007/978-3-540-30201-8_25`, `doi:10.1007/978-3-540-30201-8\_25`.

[35] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. Solving the 24-queens problem using mpi on a pc cluster. Technical Report UEC-IS-2004-6, Graduate School of Information Systems, The University of Electro-Communications, 2004. URL: `http://www.arch.cs.titech.ac.jp/~kise/doc/paper/uec-is-2004-06.pdf`.

[36] Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. In Biere et al. [7], chapter 11, pages 339–401. `doi:10.3233/978-1-58603-929-5-339`.

[37] Oliver Kullmann. Fundaments of branching heuristics. In Biere et al. [7], chapter 7, pages 205–244. `doi:10.3233/978-1-58603-929-5-205`.

[38] T.K. Satish Kumar. A model counting characterization of diagnoses. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, pages 70–76. AAAI Press, 2002. URL: `https://apps.dtic.mil/dtic/tr/fulltext/u2/p012696.pdf`.

[39] Joel Lebowitz and Giovanni Gallavotti. Phase transitions in binary lattice gases. *Journal of Mathematical Physics*, 12:1129–1133, 1971. `doi:10.1063/1.1665708`.

[40] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 366–371, 1997. URL: `http://ijcai.org/Proceedings/97-1/Papers/057.pdf`.

[41] Francois J.E. Lionnet. Question 963. *Nouvelles Annales de Mathmatiques*, 28:560, 1869.

[42] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-dnnf compilation with sharpsat. In *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, pages 356–361, 2012. URL: `https://doi.org/10.1007/978-3-642-30353-1_36`, `doi:10.1007/978-3-642-30353-1\_36`.

[43] Peter Nightingale, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Exploiting short supports for generalised arc consistency for arbitrary constraints. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 623–628, 2011. URL: `https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-111`, `doi:10.5591/978-1-57735-516-8/IJCAI11-111`.

[44] Pekka Orponen. Dempster's rule of combination is #p-complete. *Artif. Intell.*, 44(1-2):245–253, 1990. URL: `https://doi.org/10.1016/0004-3702(90)90103-7`, `doi:10.1016/0004-3702(90)90103-7`.

[45] George Pólya. Uber die doppelt-periodischen lösungen des n-damen-problems. *W. Ahrens, Mathematische Unterhaltungen und Spiele*, 1:364–374, 1921.

[46] Kevin Pratt. Closed-form expressions for the n-queens problem and related problems. *International Mathematics Research Notices*, 2019(4):1098–1107, February 2019. Advance Access Publication July 12, 2017. `doi:10.1093/imrn/rnx119`.

[47] Thomas Preußer. A brute-force solution to the 27-queens puzzle using a distributed computation. In *Membrane Computing - 19th International Conference, CMC 2018, Dresden, Germany, September 4-7, 2018, Revised Selected Papers*, pages 23–29, 2018. URL: `https://doi.org/10.1007/978-3-030-12797-8_3`, `doi:10.1007/978-3-030-12797-8\_3`.

[48] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983. URL: https://doi.org/10.1137/0212053, doi:10.1137/0212053.

[49] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 557–571, 2004. URL: https://doi.org/10.1007/978-3-540-30201-8_41, doi:10.1007/978-3-540-30201-8_41.

[50] Igor Rivin and Ramin Zabih. A dynamic programming solution to the n-queens problem. *Information Processing Letters*, 41(5):253–256, April 1992. doi:10.1016/0020-0190(92)90168-U.

[51] Günter Rote. Path problems in graphs. *Computing Supplementum*, 7:155–189, 1990. URL: http://page.mi.fu-berlin.de/rote/Papers/postscript/Path+problems+in+graphs.ps.

[52] Dan Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302, 1996. URL: https://doi.org/10.1016/0004-3702(94)00092-1, doi:10.1016/0004-3702(94)00092-1.

[53] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL: http://www.satisfiability.org/SAT04/programme/21.pdf.

[54] Tian Sang, Paul Beame, and Henry A. Kautz. Heuristics for fast exact model counting. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pages 226–240, 2005. URL: https://doi.org/10.1007/11499107_17, doi:10.1007/11499107_17.

[55] Bart Selman and Henry A. Kautz. Knowledge compilation and theory approximation. *J. ACM*, 43(2):193–224, 1996. URL: https://doi.org/10.1145/226643.226644, doi:10.1145/226643.226644.

[56] Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976. URL: https://press.princeton.edu/titles/2439.html.

[57] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[58] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA,*

August 12-15, 2006, Proceedings, pages 424–429, 2006. URL: `https://doi.org/10.1007/11814948_38`, `doi:10.1007/11814948\_38`.

[59] Seinosuke Toda. On the computational power of PP and +p. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 514–519, 1989. URL: `https://doi.org/10.1109/SFCS.1989.63527`, `doi:10.1109/SFCS.1989.63527`.

[60] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991. URL: `https://doi.org/10.1137/0220053`, `doi:10.1137/0220053`.

[61] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979. URL: `https://doi.org/10.1016/0304-3975(79)90044-6`, `doi:10.1016/0304-3975(79)90044-6`.

[62] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979. URL: `https://doi.org/10.1137/0208032`, `doi:10.1137/0208032`.

[63] Matt Vernooy and William S. Havens. An examination of probabilistic value-ordering heuristics. In *Advanced Topics in Artificial Intelligence, 12th Australian Joint Conference on Artificial Intelligence, AI '99, Sydney, Australia, December 6-10, 1999, Proceedings*, pages 340–352, 1999. URL: `https://doi.org/10.1007/3-540-46695-9_29`, `doi:10.1007/3-540-46695-9\_29`.

[64] Wei Wei and Bart Selman. A new approach to model counting. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pages 324–339, 2005. URL: `https://doi.org/10.1007/11499107_24`, `doi:10.1007/11499107\_24`.