SATISFIABILITY AND SELF-DUALITY OF MONOTONE BOOLEAN FUNCTIONS

by

Daya Ram Gaur

B. Tech., Institute of Technology, Banaras Hindu University, 1990
M. Sc., Simon Fraser University, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy in the School of Computing Science

© Daya Ram Gaur 1999
SIMON FRASER UNIVERSITY
January 1999

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-51863-9
Abstract

The problem of determining if a given monotone boolean function is self-dual arises in the areas of artificial intelligence, databases, boolean circuits, graph theory, digital signal analysis, graph theory to name a few. The problem has received considerable attention and the exact complexity of the problem is open to the best of our knowledge.

One of the initial conjectures that the problem is Co-NP-Complete was answered in the negative by Fredman and Khachiyan [19] who demonstrated the existence of a $O(n^{40(\log n)}+O(1))$ for solving the problem. Recent attempts at proving that the problem is in $\mathcal{P}$ have met with little success. The exact relationship with other problems such as graph isomorphism is also not known. Furthermore no non-trivial lower bounds are known on the running times of the algorithms for this problem.

In this thesis we formulate and study a special type of the Not All Equal Satisfiability Problem (NAESPI) which is equivalent to self-duality. We exhibit polynomial time algorithms for solving several restricted versions of NAESPI which arise naturally in differing application domains. We describe a simple $O(n^{2\log n+2})$ algorithm for NAESPI problem whose performance can be improved to $O(n^{40(\log n)}+O(1))$ (by using the observations of Fredman and Khachiyan). We study the average case behaviour of our algorithm and show that on average the algorithm terminates in $\max\{O(n^{3.87}), O(n^{3\log^2 n}), O(\frac{1}{p} - \frac{1}{\sqrt{p}})^{2+o(\log \frac{1}{p} - \frac{1}{\sqrt{p}})}\}$ time, where $p$ is the probability with which the instance is generated. We also describe an approximation algorithm for a generalization of the problem (which corresponds to a generalization of the MAX-CUT problem and is equivalent to hypergraph 2-coloring).
“Our life is frittered away by detail. Simplify, simplify.”

— Henry David Thoreau
Acknowledgments

The author would like to express his gratitude to the following people. Dr. Geir Hasle for hosting the author at SINTEF where the foundations of this work were laid and the discussions during the incipient stages of the research. Dr. Bill Havens for the excellent research environment, the encouragement without which this thesis would not be in the state it is in. Dr. Tiko Kameda for introducing the problem, numerous discussions, suggestions and careful reading of the proofs. Dr. Ramesh Krishnamurti for the denumerable hours of discussions in classroom which lead to the results in this thesis, who painstakingly ploughed through the material verifying correctness of the claims and help improve the readability of the thesis. Dr. Kaz Makino for the fruitful discussions, comments on the proofs and pointers to several related works.

The author would also like to thank Dr. Roman Bacik, Dr. Ken Jackson and Morten Irgens. Financial support in the form of research assistantships and scholarships from Dr. Havens, Dr. Krishnamurti and the School of Computing Science is gratefully acknowledged.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Quotation</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Preliminaries</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Applications</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Relational databases</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Distributed Systems</td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Model Based Diagnosis</td>
<td>6</td>
</tr>
<tr>
<td>1.2.4 Hypergraph Theory</td>
<td>7</td>
</tr>
<tr>
<td>1.2.5 Digital Signal Processing</td>
<td>8</td>
</tr>
<tr>
<td>1.2.6 Pattern Recognition and Classification</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Definitions</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 Uniform NAESPI</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2 $c$-bounded NAESPI</td>
<td>11</td>
</tr>
<tr>
<td>1.4 Results</td>
<td>13</td>
</tr>
<tr>
<td>1.5 Conclusion</td>
<td>14</td>
</tr>
</tbody>
</table>

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 NAESPI and Self-duality</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Self-duality of Monotone Boolean Functions</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Easily obtainable solution types</td>
<td>19</td>
</tr>
<tr>
<td>2.4 NAESPI and Almost Self-Dual functions</td>
<td>21</td>
</tr>
<tr>
<td>2.5 NP-Completeness</td>
<td>27</td>
</tr>
<tr>
<td>2.5.1 Finding strong solutions to NAESPI is NP-Complete</td>
<td>27</td>
</tr>
<tr>
<td>2.6 The constant case</td>
<td>29</td>
</tr>
<tr>
<td>2.6.1 3-NAESPI</td>
<td>29</td>
</tr>
<tr>
<td>2.6.2 k-NAESPI</td>
<td>31</td>
</tr>
<tr>
<td>2.7 Linear Time algorithm for solving k-NAESPI</td>
<td>37</td>
</tr>
<tr>
<td>2.8 Note</td>
<td>39</td>
</tr>
<tr>
<td>2.9 Conclusion</td>
<td>39</td>
</tr>
<tr>
<td>3 The bounded case</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>41</td>
</tr>
<tr>
<td>3.2 c-bounded k-NAESPI</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1 l-bounded k-NAESPI</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2 2-bounded k-NAESPI</td>
<td>45</td>
</tr>
<tr>
<td>3.2.3 c-bounded k-NAESPI</td>
<td>46</td>
</tr>
<tr>
<td>3.3 c-bounded NAESPI</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1 l-bounded NAESPI</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2 2-bounded NAESPI</td>
<td>50</td>
</tr>
<tr>
<td>3.3.3 c-bounded NAESPI</td>
<td>51</td>
</tr>
<tr>
<td>3.4 Bounded number of pairs of clauses</td>
<td>53</td>
</tr>
<tr>
<td>3.5 Conclusion</td>
<td>54</td>
</tr>
<tr>
<td>4 A Quasi-polynomial time algorithm</td>
<td>55</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>55</td>
</tr>
<tr>
<td>4.2 Fredman and Khachiyan’s approach</td>
<td>55</td>
</tr>
<tr>
<td>4.3 Dichotomy theorem for satisfiability</td>
<td>57</td>
</tr>
<tr>
<td>4.4 Algorithm</td>
<td>59</td>
</tr>
<tr>
<td>4.4.1 Read-k boolean formulas</td>
<td>63</td>
</tr>
</tbody>
</table>
4.5 Variations of the algorithm ............................................. 63
4.6 Conclusion ................................................................. 66
4.7 Note ........................................................................... 67

5 Average Case Analysis ..................................................... 68
  5.1 Introduction ............................................................. 68
  5.2 Random NAESPI ......................................................... 69
  5.3 A Variant ................................................................. 73
  5.4 Analysis ................................................................. 75
  5.5 Conclusion ............................................................. 76

6 Approximation Algorithm for NAESP ................................. 77
  6.1 Introduction ............................................................. 77
  6.2 Approximation Algorithms ............................................ 80
    6.2.1 \( \frac{1}{2} \) Approximation Algorithm .......................... 80
    6.2.2 \( \frac{5}{4+1} \) Approximation Algorithm ...................... 81
  6.3 Conclusion ............................................................. 83

7 Conclusion and Open problems ......................................... 84

A List of Symbols ............................................................ 86

Bibliography ........................................................................ 91
List of Figures

3.1 auxiliary graph for the projective plane NAESPI .................. 43
3.2 1-bounded NAESPI ................................................. 44
Chapter 1

Preliminaries

1.1 Introduction

In this thesis we address the worst case and the average case complexity of the problem of determining if a given monotone boolean function is self-dual. We also look at some restricted versions of the above problem. A problem \( \Pi \) is said to be polynomially solvable (or in class \( \mathcal{P} \)) if there exists an algorithm \( A \) such that \( A \) solves every instance of \( \Pi \) in time polynomial in the length of the input. The problem \( \Pi \) is said to be non-deterministically polynomially solvable (or in class \( \mathcal{NP} \)) if there exists a non-deterministic algorithm \( A \) such that \( A \) solves every input instance of \( \Pi \) in time polynomial in the length of the input. A problem \( \Pi \) is called NP-Hard if every problem in NP is polynomially reducible to \( \Pi \). Problem \( \Pi \) is called NP-Complete if it is in NP and NP-Hard. Co-NP-Complete is the class of problems whose complement is NP-Complete. The work of Edmonds [12] contains an informal definition of \( \mathcal{NP} \), a notion which was later formalized by Cook [10]. Cook also showed that the problem of deriving an assignment which satisfies a given set of clauses (satisfiability) was NP-Complete. Later Karp [31] reduced a host of other problems to the satisfiability problem to show that they are NP-Complete. One of the outstanding open questions in the theory of computational complexity is whether \( \mathcal{P} = \mathcal{NP} \). It is widely believed that \( \mathcal{P} \neq \mathcal{NP} \). Hence, problems which are not known to be NP-Complete or do not have known polynomial time algorithms are of great interest to the community. The graph isomorphism problem [21] is a classic example of such a problem; self-duality of monotone
A potential criticism of the worst case analysis of an algorithm is that the exponential behaviour of the algorithm might be caused by a small set of instances which do not occur frequently. This problem could be tackled by analyzing the algorithm for the expected (average) time. In this thesis we study the worst case and the average case complexity of algorithms for the problem of self-duality of monotone boolean functions and some related problems. We show that several special cases of the problem can be solved in polynomial time. The general problem is showed to be solvable in quasi-polynomial time.

The problem of determining if a given monotone boolean function is self-dual arises in the areas of artificial intelligence, databases, boolean circuits, graph theory, and digital signal analysis, to name a few. The problem has received considerable attention [5, 4, 3, 14, 7, 28, 26, 38, 19] and the exact complexity of the problem is open to the best of our knowledge.

One of the initial conjectures that the problem is Co-NP Complete was answered in the negative by Fredman and Khachiyan [19], who demonstrated the existence of a $n^{4\alpha(\log n) + O(1)}$ for solving the problem. Recent attempts at proving that the problem is in $\mathcal{P}$ have met with little success. The exact relationship with other problems such as graph isomorphism is also not known. Furthermore no non-trivial lower bounds are known on the running times of the algorithms for this problem.

Problems in the area of probabilistic knowledge representation, which are equivalent to self-duality, have been identified by Khardon [33] and Kavvadias, Papadimitriou and Sideri [32]. Bioch and Ibaraki [3] describe a host of problems which are equivalent to determining the self-duality of monotone boolean functions. They also address the question of the existence of incrementally polynomial algorithms for solving the problem of determining the self-duality of monotone boolean functions, where an algorithm to enumerate items $a_1, a_2, \ldots, a_k$ is incrementally polynomial [30, 35], if the time taken to output the $i^{th}$ item $a_i$ is polynomial in the sizes of items $a_1, a_2, \ldots, a_i-1$.


---

1It is widely believed that NP-Complete and Co-NP Complete problems are not solvable in $n^{4\alpha(\log n) + O(1)}$ time.
of \( n \) variables. It has been shown that for \( 2\)-monotone [7] boolean functions, it is possible to check the self-duality in polynomial time. Bioch and Ibaraki [5] define \textit{almost self-dual} functions as an approximation to the class of self-dual functions. They describe an algorithm based on almost self-duality to determine if a function is self-dual. The complexity of their procedure is exponential in the worst case. Ibaraki and Kameda [28] show that every self-dual function can be decomposed into a set of majority functions over three variables. This characterization in turn gives an algorithm (though not polynomial) for checking self-duality. Makino and Ibaraki [38] define the latency of a monotone boolean function and relate it to the complexity of determining if a function is self-dual. Makino [37] investigates several classes of positive and horn boolean functions and exhibits either polynomial time algorithms or the NP-Completeness of these classes.

In this chapter we describe several applications of the problem of determining self-duality of monotone boolean functions [14]. We recast the problem as a special type of satisfiability problem and discuss several natural restrictions of the problem arising in various application domains which merit some detailed study. Section 1.2 describes the applications. Section 1.3 describes some of the basic definitions and the restrictions arising naturally in the application domains under which we study the problem in some detail in this thesis. Section 1.4 is a summary of the results presented in this thesis.

1.2 Applications

Eiter and Gottlob [14] describe some applications of the problem; we summarize their work in this section.

1.2.1 Relational databases

One of the principal design goals for a relational database system is to have a set of schemes which allows the storage of information without any unnecessary redundancy. Such schemes can be constructed by using some \textit{normal forms} which satisfy certain constraints on the data. One such type of constraint is the functional dependency constraint (to be defined later).

To facilitate the discussion we will introduce some terms. A \textit{domain} is simply a set of values. A \textit{relation} is a subset of the Cartesian product of one or more domains. The
members of a relation are called *tuples*. A relation can be visualized as a table, where each row is a tuple and each column corresponds to a domain. The columns are also referred to as *attributes* at times. If \( t \) is a tuple and \( X \) a set of attributes (domains), \( t[X] \) denotes the components of \( t \) in the attributes of \( X \), we are given a set of attributes \( R \), and \( A \subseteq R \) and \( B \subseteq R \).

**Definition 1.1 (Functional dependency)** The functional dependency \( A \rightarrow B \) holds on \( R \) if for every relation \( r(R) \), and for all pairs of tuples \( t_1, t_2 \):

\[
t_1[A] = t_2[A] \Rightarrow t_1[B] = t_2[B].
\]

Given a set of functional dependencies \( F \) over a relation \( R \) we denote the closure of \( F \) by \( F^+ \) which is obtained by using the standard assumptions of reflexivity, transitivity and symmetry.

**Definition 1.2 (Armstrong Relation)** A relation is called an Armstrong relation if its functional dependencies satisfies \( F = F^+ \).

It has been advocated by Mannila and Räihä [39] that Armstrong relations are a good tool for database design. Hence, the question of determining whether a given set of functional dependencies over a relation is equal to the closure of the functional dependencies is an interesting problem for the database designer. To state the problem formally, we need the concept of *Boyce-Codd normal form*.

**Definition 1.3 (Super key)** \( K \subseteq R \) is called a super key if for any relation \( r(R) \), for all pairs of tuples \( t_1, t_2 \) such that \( t_1 \neq t_2 \), \( t_1[K] \neq t_2[K] \).

We require this definition below in the definition of Boyce-Codd normal form.

**Definition 1.4 (Boyce-Codd normal form)** A relation \( r(R) \) is in Boyce-Codd normal form if for all functional dependencies \( X \rightarrow Y \), at least one of the following holds:

- \( X \rightarrow Y \) is a trivial functional dependency (that is \( Y \in X \)),
- \( X \) is a super key of \( R \).
CHAPTER 1. PRELIMINARIES

We now define the problem of determining if a given relation \( r(R) \) together with a set of functional dependencies is an Armstrong relation.

**PROBLEM: FD RELATIONAL EQUIVALENCE**

**INSTANCE:** A relation \( r(R) \) and a set of functional dependencies in Boyce-Codd normal form.

**QUESTION:** Is \( R \) an Armstrong relation?

Eiter and Gottlob [14] showed that the above problem is polynomially equivalent to the problem of determining if a hypergraph is saturated (we define the hypergraph saturation problem later). In Chapter 2, we establish the equivalence of the not all equal satisfiability problem and the hypergraph saturation problem.

### 1.2.2 Distributed Systems

Mutual exclusion guarantees safe execution of critical operations in distributed systems. Mutual exclusion is typically achieved by means of coteries [2].

**Definition 1.5 (Coterie)** A coterie \( C = \{Q_1, Q_2, \ldots, Q_n\} \) is defined as a collection of subsets (quorums) \( Q_i \subseteq \{1, \ldots, m\}, i = 1 \ldots n \) such that every pair of subsets has a non-empty intersection and \( \forall i \in \{1, \ldots, n\} \ Q_i \nsubseteq Q_j \) for some \( j \in \{1, \ldots, n\} \).

A group of sites can perform a critical operation only if it contains a quorum from the set \( C \). We need consider only minimal quorums hence the maximality assumption (second assumption) in the definition of coterie.

**Definition 1.6 (Non-dominated coterie)** A coterie \( C \) is called non-dominated (ND) if there does not exist any other coterie \( C' \) such that \( \forall Q \in C, \text{ there is a } Q' \in C' \text{ for which } Q' \subseteq Q \).

**ND** coteries are of interest because of reliability considerations [2]. The following problem is polynomially equivalent to the problem of determining if a monotone boolean function is self-dual.

**PROBLEM:** ND Coterie

**INSTANCE:** A coterie \( C \).

**QUESTION:** Is \( C \) non-dominated?
1.2.3 Model Based Diagnosis

Diagnostic reasoning, a technique proposed by deKleer and Williams [11] and studied by Reiter [43], attempts to identify the dependencies between components of a model and the discrepancies in the observations. Suppose we have a model for some circuit and the output is not correct for a particular input. Diagnosis entails identifying the minimal set of components such that failure of any one component in the set could have caused the malfunction. Such a set is called a conflict set. A conflict set is minimal if it does not contain any other conflict set. A model can now be described as a disjunction of minimal conflict sets, where each conflict set is a conjunction. For example, let $M$ be a circuit with three components $\{m_1, m_2, m_3\}$. One of the conflict sets could be $\{m_1, m_2\}$ associated with some observation $c$. This means that if $c$ has been observed then either one of $m_1$ or $m_2$ is not functioning correctly. To comment on the nature and properties of conflicts we quote deKleer and Williams from [11].

For complex domains any single symptom can give rise to a large set of conflicts, including the set of all components in the circuit. To reduce the combinatorics of diagnosis it is essential that the set of conflicts be represented and manipulated concisely. If a set of components is a conflict, then every superset of that set must also be a conflict. Thus the set of conflicts can be represented concisely by only identifying the minimal conflicts, where a conflict is minimal if it has no proper subset which is also a conflict. This observation is central to the performance of our diagnostic procedure. The goal of conflict recognition is to identify a complete set of minimal conflicts.

Formally, a system is a pair $(SD, COMPONENTS)$ where, $SD$ is a set of first order sentences and $COMPONENTS$ is a finite set of constants. $SD$ contains a unary distinguished predicate called $AB()$ where $AB(x)$ is interpreted to mean that $x$ is ‘abnormal’. An observation of a system is a finite set of first order sentences denoted $OBS$.

**Definition 1.7 (Diagnosis)** A diagnosis for $(SD,COMPONENTS,OBS)$ is a minimal set $\Delta \subseteq COMPONENTS$ such that

$$SD \cup OBS \cup \{AB(c) | c \in \Delta\} \cup \{\neg AB(c) | c \in COMPONENTS - \Delta\}$$

is consistent.
In other words a diagnosis is the smallest set of components such that the assumption that each abnormal component is in the set together with the assumption that all the other components are not abnormal is consistent with the observations and the system description.

**Definition 1.8 (Conflict set)** A Conflict set is a set \( C \subseteq \text{COMPONENTS} \) such that

\[ SD \cup OBS \cup \{ \neg AB(c) | c \in C \} \]

is consistent.

Eiter and Gottlob [14] show the following theorem:

**Theorem 1.1 (Eiter and Gottlob [14])** Let \( C \) be the set of minimal conflict sets of \((SD, COMPONENTS, OBS)\) and \( D \) be a set of diagnoses for \((SD, COMPONENTS, OBS)\). Given \( C \) and \( D \) for input, deciding if there is an additional diagnoses not contained in \( D \) is polynomially equivalent to Co-SIMPLE-H-SAT.

SIMPLE-H-SAT is the simple hypergraph saturation problem defined below.

### 1.2.4 Hypergraph Theory

**Definition 1.9 (Hypergraph)** A hypergraph \( H \) is pair \((V, E)\) where \( V \) is a finite set and \( E \) is a family of finite subsets of \( V \).

A hypergraph is called simple if there are no pairs of edges \( E_i, E_j \) such that \( E_i \subseteq E_j \). A simple hypergraph is sometimes referred to as a Sperner family.

**PROBLEM:** SIMPLE-H-SAT

**INSTANCE:** A simple hypergraph \( H \).

**QUESTION:** Is \( H \) saturated? i.e., is every subset of the vertex set contained in an edge or contains an edge of the hypergraph?

Another problem in hypergraph theory which is equivalent to self-duality is TRANS-HYP. A transversal of a hypergraph \( H = (V, E) \) is a set \( V' \subseteq V \) such that \( V' \cap E_i \neq \emptyset \) for all \( E_i \in E \).

**Definition 1.10 (Transversal of a hypergraph)** The transversal hypergraph \( Tr(H) \) of a hypergraph \( H \) is the family of all minimal transversals of \( H \).
CHAPTER 1. PRELIMINARIES

PROBLEM: TRANS-HYP

INSTANCE: Two hypergraphs $H$ and $G$.

QUESTION: Is $Tr(G) = H$, i.e., is $H$ a transversal hypergraph of $G$?

1.2.5 Digital Signal Processing

Here we assume basic familiarity with digital signal processing. Digital data which is being transmitted has to be reconstructed at the receiving end due to noise. If the noise is Gaussian or additive in nature then linear filters suffice for the reconstruction of the data. If the noise is neither gaussian nor additive then linear filters are not a viable tool for data reconstruction. Also, linear filters tend to lose crucial high frequency components such as edges in a digital picture. A new class of non-linear filters called stack filters have been proposed by Wendt, Coyle and Lin [46] to get around this problem.

Stack filters can be uniquely represented using monotone boolean functions. If the stack filter identifies some vector $x$ as being noise then the complement of $x$ cannot be noise. Hence, it is in the interest of the filter designer to ensure that for every vector and its complement exactly one of them is classified as noise. This problem is equivalent to determining if a monotone boolean function is self-dual.

1.2.6 Pattern Recognition and Classification

Suppose we are to come up with a model which classifies the data into 'good' or 'bad' based on some features. Without loss of generality we can assume that the features take on boolean values. A good pattern classifier by definition should be able to classify every input pattern. Also, if an input $x$ is 'good' then its complement cannot be 'good' and vice versa. Determining if a pattern classifier has the above mentioned property is equivalent to determining if some monotone boolean function is self-dual. This problem has applications in the areas of knowledge acquisition and data mining [37].
1.3 Definitions

In this section we give some definitions which are used subsequently. Given a boolean function \( f(x_1, x_2, \ldots, x_n) \), its dual denoted by \( f^d \) is defined as follows:

**Definition 1.11 (Dual)** \( f^d(x) = \bar{f}(\bar{x}) \), for all vectors \( x = (x_1, x_2, \ldots, x_n) \in \{0, 1\}^n \).

A boolean function is monotone if the following definition holds:

**Definition 1.12 (Monotone boolean function)** A boolean function \( f \) is monotone if \( f(x) \leq f(y) \) for all vectors \( x \leq y \). A vector \( x \leq y \) if for every coordinate \( x_i \leq y_i \).

We are interested in the following problem:

**Problem: Self-Duality**

**Instance:** A monotone boolean function \( f \) in DNF.

**Question:** Is \( f \) self-dual?

Given such a function \( f \) now, we will construct another problem called the Not All Equal Satisfiability Problem with Intersection (NAESPI) and show the equivalence of the two problems. Next we define the NAESPI problem.

**Problem: NAESPI**

**Instance:** A monotone boolean function \( f \) in CNF such that every pair of clauses in \( f \) has a non-empty intersection.

**Question:** Is there a boolean vector such that every clause contains at least one variable set to 1 and at least one variable set to 0?

In this thesis we show that NAESPI is equivalent to SELF-DUALITY. We also investigate the following restrictions of NAESPI:

1. \( k\text{-}NAESPI \): each clause in the input has at most \( k \) variables.
2. uniform \( NAESPI \): each clause has the same number of variables.
3. \( c\text{-}bounded \text{ NAESPI} \): Every pair of clauses intersects in at most \( c \) variables.
4. uniform \( c\text{-}bounded \text{ NAESPI} \): instances which are \( c\text{-}bounded \) and uniform.
5. *uniform c-bounded k-NAESPI*: instances of k-NAESPI which are c-bounded and uniform.

In addition we investigate the NAESPI problem with the intersection restriction removed (called NAESP). NAESP is a generalization of the MAX-CUT problem defined below.

**PROBLEM:** MAX-CUT

**INSTANCE:** An undirected graph $G = (V, E); k \in \mathcal{N}$.

**QUESTION:** A partition of $V$ into $S$ and $V - S$ such that the number of edges from $S$ and $V - S$ is at least $k$.

We describe a simple approximation algorithm for the NAESP problem in this thesis.

It should be noted that c-bounded NAESPI subsumes c-bounded $k$-NAESPI but we investigate the latter using a completely different approach. It might appear that these restriction are artificial at first sight. We argue below that these restrictions arise in various application domains. Also, as advocated by Johnson [29], a good approach to resolving the complexity of a problem is to study the problem under various restrictions, hopefully gaining enough insight to tackle the general case.

Coteries, originally proposed by Lamport for achieving 'mutual exclusion' in distributed systems [34], is a general mechanism which can also be used to implement coordinator election [20], and modular redundancy systems for fault-tolerant computing [45]. Coteries when used to implement mutual exclusion can be thought of as an explicit enumeration of all the sets of nodes which can perform the restricted operation simultaneously. Elements of the set \{1, \ldots, m\} over which the quorums are defined are referred to as nodes at times in this chapter.

Reliability is an important issue in a distributed system. We assume that under failure if some quorum is the sole survivor then the nodes in the quorum are still connected. This assumption simplifies the topology issues associated with the underlying network. Failures can be analyzed either deterministically or probabilistically. Barbara and Garcia-Molina [2] describes a deterministic measure of reliability called node vulnerability of a coterie. Next, we argue that "uniform" coteries are desirable due to reliability considerations in a distributed system.
1.3.1 Uniform NAESPI

Definition 1.13 (Node Vulnerability) Node vulnerability of a coterie $C$ is the size of the smallest subset $V$ of nodes in $C$ such that $V$ has non-empty intersection with every quorum in $C$.

From the definition of node vulnerability the following theorem is evident.

Theorem 1.2 [Barbara and Garcia-Molina [2]] If $S$ is a non-dominated coterie, then its node vulnerability is the cardinality of the smallest quorum in $S$.

Theorem 1.2 implies that the best choices as far as node vulnerability is considered for non-dominated coteries are uniform coteries with quorum sizes $\frac{n+1}{2}$ for odd $n$ and $\frac{n}{2} + 1$ or $\frac{n}{2}$ for even $n$. Hence Barbara and Garcia-Molina [2] define uniform coteries.

Definition 1.14 (Uniform coteries) A coterie is $k$-uniform if it is non-dominated and all its quorums are of size $k$. A $k$-uniform coterie for some $k$ is called uniform coterie.

Given that we are interested in finding non-dominated coteries with the highest node vulnerability, we are interested in determining if a given uniform coterie is non-dominated. Hence it is natural to study the NAESPI problem under the uniformity restriction.

1.3.2 $c$-bounded NAESPI

Maekawa [36] describes an algorithm that uses $c\sqrt{N}$ messages to achieve mutual exclusion in a distributed network with $N$ nodes for some constant $c$. It is also known that $3\sqrt{N}$ is a lower bound on the number of messages needed to achieve mutual exclusion. The algorithm of Maekawa [36] relies on coteries with the additional properties listed below. In what follows, a coterie is denoted $C$ and the $i^{th}$ quorum in $C$ is denoted $Q_i$. $N$ is the number of nodes over which the coterie is defined.

1. $Q_i$, $1 \leq i \leq N$, always contains $i$.
2. Each $Q_i$ is of size $K$.
3. Any node $j \in \{1 \ldots N\}$ belongs to the same number of quorums.
CHAPTER 1. PRELIMINARIES

The first property simply reduces the number of messages sent or received by any node. The second property ensures that all the nodes receive and send the same number of messages to achieve mutual exclusion. The third property states that each node serves as an arbitrator for the same number of nodes. If \( K \) is a power of some prime \( p \) then such a coterie corresponds to a finite projective plane of order \( p \) [1]. For other values of \( K \), Maekawa [36] describes some methods for generating coteries with the desired properties. We describe another method to obtain coteries with the properties stated above.

Let graph \( G_n \) be a union of \((n + 1)\) cliques \( \{Q_1, Q_2, \ldots, Q_{n+1}\} \) of size \( n \) such that every two cliques intersect in exactly one vertex. We define \( C = \{Q_1, Q_2, \ldots, Q_{n+1}\} \) to be a coterie. Since \( G_n \) obeys the minimality and intersection property, \( C \) is a coterie. \( G_n \) can be constructed in the following fashion. Vertices of \( G_n \) are all the 2-subsets of the set \( \{1, 2, \ldots, n\} \). There is an edge between two vertices if the corresponding sets have a non-empty intersection. It is easy to see that \( G_n \) so constructed has \((n + 1)\) cliques of size \( n \). The total number of nodes in \( G_n \) is \( (\binom{n}{2}) \). We now need to assign a quorum to every vertex in \( G_n \). As there are only \((n + 1)\) quorums but \( (\binom{n}{2}) \) vertices, each quorum would be assigned to \( q = (\binom{n}{2})/(n + 1) \) vertices.

The quorums of a coterie derived using the mechanism outlined above for \( K = 5 \) are shown below:

\[
\begin{align*}
\{1, 2, 3, 4, 5\} \\
\{1, 6, 7, 8, 9\} \\
\{2, 6, 10, 11, 12\} \\
\{3, 7, 10, 13, 14\} \\
\{4, 8, 11, 13, 15\} \\
\{5, 9, 12, 14, 15\}
\end{align*}
\]

In such a coterie \( K = \sqrt{2 \times N} - 1 \). Therefore the number of messages is at most \( 2K = 2\sqrt{2\sqrt{N}} \) versus the lower bound of \( 3\sqrt{N} \) [36]. Furthermore, each \( j \in \{1 \ldots N\} \) belongs to at most 2 quorums. Also, \( |Q_i \cap Q_j| = 1 \) for all \( i \neq j \in \{1 \ldots N\} \). This coterie is non-dominated but is almost optimal in terms of the number of messages needed to achieve mutual exclusion. The relationship between the non-domination of coteries and the
minimum number of messages needed to attain mutual exclusion has not been studied to
the best of our knowledge.

We argue that the class of coterries presented above is almost optimal in the number
of messages needed to achieve mutual exclusion. Furthermore these coterries have bounded
intersection in the clauses. That is, for any two quorums \( Q_i, Q_j \), \( |Q_i \cap Q_j| \leq c \) for some
constant \( c \). Hence it might be a good idea to determine the exact complexity of the
\textit{c-bounded-uniform-NAESPI} problem, where \textit{c-bounded-uniform-NAESPI} is equivalent to the
\textit{c-bounded-uniform-coterries}. It would also be fruitful to study a generalization of the
\textit{c-bounded-uniform NAESPI} in which the uniformity condition is dropped (the class \textit{c-bounded
NAESPI}). In the next section we outline our contributions.

\section{Results}

In this thesis we first establish the equivalence between \textit{NAESPI} and \textit{Self-duality}. We
describe an \( O(n^2 \log n + 2) \) algorithm for solving the \textit{NAESPI} problem. Our bound of \( O(n^2 \log n + 2) \)
is strictly worse than the running time of the algorithm proposed by Fredman and Khachiyan
\cite{19} which is \( O(n^{4\alpha \log n} + O(1)) \), but our algorithm is simple, with a simpler analysis of the
running time and the approach is different from the approach of Fredman and Khachiyan.
We also show that an adaption of our algorithm using the technique of Fredman and
Khachiyan runs in \( O(n^{4\alpha \log n} + O(1)) \) time. We also show that the bound is tight for the
given algorithm and provide an average case analysis of an adaptation of our algorithm for
a randomly generated instance of \textit{NAESPI}. The average case running time is

\[
\max\{O(n^{3.87}), O(n^3 \log^2 n), O\left(\frac{1}{p} - \frac{1}{\sqrt{p}}\right)^2 + o\left(\frac{1}{p}\right)\},
\]

where \( p \) is the probability with which the instance is generated. We show that \textit{k-NAESPI}
and \textit{c-bounded NAESPI} can be solved in \( O((nk)^{k+1}) \) and \( O(n^{2c+2}) \) time. We also describe
an algorithm linear in \( n \) for the \textit{k-NAESPI} problem. We show that for \textit{uniform-c-bounded
NAESPI} the number of clauses \( n \) is bounded. We also show that finding special types of
solutions to the \textit{NAESPI} problem is \textit{NP-Complete}. Finally we describe an approximation
algorithm for the \textit{NAESP problem} (where the intersection property is relaxed).

The organization of the thesis is as follows, Chapter 2 shows the equivalence between
\textit{NAESPI} and \textit{Self-duality}, and provides an \( O((nk)^{k+1}) \) algorithm for solving the \textit{k-NAESPI}
1. PRELIMINARIES

problem. The algorithm is then improved to an algorithm, that is linear in the number of clauses $n$. It also shows that finding a 'particular' type of solutions to NAESPI is NP-Complete. We also give an alternate characterization of almost self-dual functions in terms of NAESPI instances which do not admit easy solutions.

Chapter 3 shows that for $c$-bounded $k$-NAESPI, the number of clauses is bounded. This bound on the number of clauses implies that $c$-bounded $k$-NAESPI can be solved in $O(n^{c+1}k)$ time for some constants $c$ and $k$. We show that $c$-bounded NAESPI can be solved in $O(n^{2c+2})$ time. As $c$-bounded $k$-NAESPI is a subclass of $c$-bounded NAESPI the latter result is weaker.

Chapter 4 describes an $O(n^{2\log n+2})$ algorithm for solving NAESPI. We show that the analysis is tight and rule out some obvious modifications which could be made to the algorithm to improve its complexity.

Chapter 5 describes a way of generating NAESPI instances using random graphs. We describe a variant of our basic algorithm for solving NAESPI. This variant is based on the observations of Fredman and Khachiyan [19]. We show that the new algorithm terminates in $O(n^{4\log n+O(1)})$ time. We also present an average case analysis for the new algorithm, given that the input instances are generated in a random way. The average case running time is

$$\max\{O(n^{3.87}), O(n^3 \log^2 n), O\left(\frac{1}{p} - \frac{1}{\sqrt{p}}\right)^{2+o(\log \frac{1}{p})}\},$$

where $p$ is the probability with which the instance is generated.

Chapter 6 describes a $\frac{s}{s+1}$ approximation algorithm for the $NAESP$, where $s$ is the minimum number of variables in each clause.

1.5 Conclusion

In this chapter we introduced the problem of determining if a monotone boolean function is self-dual. We identified several applications of the problem. We also defined the Not All Equal Satisfiability Problem with Intersection (NAESPI) and several restrictions to it. We discussed the relevance of the restricted versions of NAESPI to the application areas. Finally, we outlined the contributions made in this thesis. We hope that the results presented in this thesis would lead to an improved understanding of issues related to the problem of
CHAPTER 1. PRELIMINARIES

self-duality.
Chapter 2

NAESPI and Self-duality

2.1 Introduction

In this chapter we introduce the problem of determining whether a monotone boolean function is self-dual. We also introduce the not all equal satisfiability problem with only positive literals (NAESP). Next we show that self-duality of monotone boolean functions is equivalent to the satisfiability of a special type of NAESP problem, which we refer to as the NAESPI problem. Having established the equivalence of the two problems, we will identify two special cases of NAESPI which can be solved in polynomial time. One of these is tantamount to restricting the definition of a solution to the NAESPI. We study the relationship of this class of NAESPI instances to the class of almost self-dual functions defined by Bioch and Ibaraki [5]. For another type of restriction on the solution to NAESPI, we show that the problem is NP-Complete. We then describe an $O((nk)^{k+1})$ algorithm for solving the $k$-NAESPI problem. Finally, we improve the running time algorithm of the algorithm for the $k$-NAESPI problem to $O((2^k)^{kn})$, which is linear in $n$.

2.2 Self-duality of Monotone Boolean Functions

First let us recapitulate some of the definitions presented in the previous chapter. Given a boolean function $f(x_1, x_2, \ldots, x_n)$, we define its dual denoted by $f^d$ as follows:
CHAPTER 2. NAESPÍ AND SELF-DUALITY

Definition 2.1 (Dual) \( f^d(x) = \overline{f(\overline{x})} \), for all vectors \( x = (x_1, x_2, \ldots, x_n) \in \{0, 1\}^n \).

Next we define monotone boolean functions.

Definition 2.2 (Monotone boolean function) A boolean function \( f \) is monotone if \( f(x) \leq f(y) \) for all vectors \( x \leq y \). A vector \( x \leq y \) if for every coordinate \( x_i \leq y_i \), \((0 < 1)\).

Equivalently, a boolean function is monotone if it has a DNF without any negative literals. If a monotone function \( f \) is in disjunctive normal form (DNF) then \( f^d \) can be obtained by interchanging every and operator with an or operator and vice versa. \( f^d \) will then be in conjunctive normal form (CNF).

Self-Duality can now be defined as:

PROBLEM: Self-Duality.

INSTANCE: A boolean function \( f(x_1, x_2, \ldots, x_n) \).

QUESTION: Is \( f^d(x) = f(x) \) for every vector \( x \in \{0, 1\}^n \).

From the definition of self-duality it follows that:

Property 1 A boolean function \( f \) is self-dual \( \iff \) for all vectors \( x \in \{0, 1\}^n \), \( f(x) \neq f(\overline{x}) \).

Determining if an arbitrary boolean function \( f \) (not necessarily monotone) is self-dual is Co-NP Hard. However, whether a monotone function is self-dual can be determined in polynomial time is still open. Self-duality of monotone boolean functions was believed to be a strong contender for a Co-NP Complete problem [20] until Fredman and Khachiyan [19] discovered a \( n^{4\sqrt{\log(n)} + O(1)} \) algorithm for the problem.

We can assume that the monotone function \( f \) whose self-duality we are interested in is in disjunctive normal form. Given a DNF representation of the monotone function \( f \), we can think of \( f \) as being comprised of terms, where all the variables in a term are connected by the and operator and all the terms are connected by the or operator.

Next we show that if there exists a pair of terms in monotone function \( f \) which do not intersect in any variable, then \( f \) is not self-dual. This observation is also implicit in [19].

Lemma 2.1 If there exists a pair of non-intersecting terms in a monotone function \( f \) then \( f \) is not self-dual.
CHAPTER 2. NAESPI AND SELF-DUALITY

Proof: Let $T_1$ and $T_2$ be two such terms. Let $x$ be a \{0, 1\}" vector in which all the variables in $T_1$ are set to 1 and the variables in $T_2$ are set to 0. The remaining variables are arbitrarily set to 0 or 1. $f(x) = 1$ as the term $T_1$ evaluates to 1. Also, $f(\bar{x}) = 1$ as $T_2$ evaluates to 1. Hence by Property 1 $f$ is not self-dual.

Lemma 2.1 allows us to focus only on those monotone boolean functions in which every pair of terms has a non-empty intersection. Henceforth we will assume that all monotone boolean functions under consideration have this pair-wise intersection property. Such monotone boolean functions are also referred to as dual-minor functions [28].

Another assumption which we will use throughout this thesis is the following:

**Property 2** Every variable in $f$ belongs to at least two terms in $f$.

This assumption is valid because the variables which belong only to a single term can be used to satisfy the term once some variables are set to a value of 1 or 0 in the term. Property 2 coupled with Lemma 2.1 implies that each term has at most $n$ variables where $n$ is the total number of clauses in $f$. Therefore the total number of variables $m \leq n^2$ in $f$.

**PROBLEM: NAESPI**

**INSTANCE:** A collection of terms $T = (T_1, T_2, \ldots, T_n)$, $T_i \subseteq V = \{v_1, v_2, \ldots, v_m\}$ such that every pair of terms $T_i, T_j$ has a non-empty intersection.

**QUESTION:** Is there a set $S \subseteq V$ such that $S$ contains at least one element from every term and no term is contained inside $S$.

The next lemma states that the complement of a solution to a given NAESPI problem $P(f)$ is also a solution to $P(f)$.

**Lemma 2.2** If $S$ is solution to a given NAESPI problem $P(f)$ then so is $\bar{S}$.

We are interested in the self-duality of $f$ in DNF form. Let $P(f)$ be the NAESPI problem obtained by treating each term of $f$ as a clause in $P(f)$. For example, if $f = (x_1 \land x_2) \lor (x_1 \land x_3) \lor (x_2 \land x_3)$ then $P(f) = (x_1 \lor x_2) \land (x_1 \lor x_3) \land (x_2 \lor x_3)$. Every pair of clauses in $P(f)$ obeys the intersection property as $f$ obeys the intersection property. The next lemma asserts that $f$ is not self-dual if and only if $P(f)$ is satisfiable.

**Lemma 2.3** $f$ is not self-dual $\iff$ NAESPI $P(f)$ is satisfiable.
Chapter 2. NAESPI and Self-duality

Proof: ⇒ Assume that \( f \) is not self-dual. By Property 1 we have a vector \( x \) such that \( f(x) = f(\bar{x}) \). There are two cases:

- \( f(x) = f(\bar{x}) = 1 \). Let \( T_i \) be the term in \( f \) which evaluates to 1 for \( x \). For \( \bar{x} \), \( T_i \) evaluates to 0. As \( T_i \) intersects every clause in \( f \), each term in \( f \) has at least one variable set to 0. This is a contradiction as \( f(\bar{x}) = 1 \). Hence this case cannot happen. This also amounts to saying that the function is not dual-minor, hence it cannot be self-dual.

- \( f(x) = f(\bar{x}) = 0 \). Each term in \( f \) contains at least one 0 because \( f(x) = 0 \). Similarly each term in \( f \) contains at least one 1 as \( f(\bar{x}) = 0 \). Let \( S \) be the union of all the variables in \( f \), which are assigned to 1 in the vector \( x \). Thus \( S \) contains at least one element from each term in \( P(f) \) and does not contain at least one element from each term in \( P(f) \). Hence \( S \) intersects every clause in \( P(f) \) but does not contain any clause in \( P(f) \). Therefore, \( S \) is a valid solution.

⇐ Given a solution \( S \) to \( P(f) \), let \( x \in \{0,1\}^n \) be the vector constructed in the following way:

\[
x_i = 1 \text{ if } x_i \in S \text{ else } x_i = 0
\]

Clearly, \( f(x) = 0 \). Since \( \bar{S} \) is also a solution to \( P(f) \) (by Lemma 2.2), taking the characteristic vector \( x \) of \( \bar{S} \) implies \( f(\bar{x}) = 0 \). Hence by Property 1 \( f \) is not self-dual.

In this section we have established the equivalence of NAESPI and the complement of self-duality of monotone boolean functions. In the next section we describe two particular types of solutions to NAESPI which can be computed in polynomial time.

2.3 Easily obtainable solution types

Definition 2.3 (Easy solution) Given an NAESPI problem \( P(f) \), let \( S \) be a solution such that \( S \) is contained in some clause of \( P(f) \). We call \( S \) an easy solution to \( P(f) \).

Definition 2.4 (Easily satisfiable) An NAESPI problem is easily satisfiable if it admits an easy solution.
CHAPTER 2. NAESPI AND SELF-DUALITY

Given an easy solution $S$ to the NAESPI problem $P(f)$, we show that either there exists a clause $C \in P(f)$ such that $C$ intersects $S$ in all but one variable from $C$ or we can augment $S$ until the above mentioned property is true. Armed with this fact we can try out all the possible valid subsets to see if any one of them is a solution. As the number of valid subsets is polynomial in $n$, we terminate in time polynomial in $n$. More formally we need the following lemma:

**Lemma 2.4** Let $S$ be an easy solution to the NAESPI problem $P(f)$. $S$ can be extended to another easy solution $S'$ such that for a clause $C \in P(f)$, $|C \cap S'| = |C| - 1$.

**Proof:** Let $C_0$ be the clause which contains $S$. Let $a$ be an element of $C_0$ not in $S$. Let $S' = S \cup \{a\}$. If $S'$ is not a solution to $P(f)$, then there is some clause $C = S'$, in which case $|C \cap S| = |C| - 1$. We keep adding elements to $S$ iteratively.

Lemma 2.4 implies that if for every clause $C$, we try all the $|C|$ subsets of $C$ of size $|C| - 1$ we should obtain our easy solution if one exists. Let $n$ be the number of clauses in $P(f)$. It takes $O(n^2)$ time to verify if a given subset is a solution to $P(f)$. As there are at most $n \times |C| \leq n \times n$ subsets which have to be tried, we can find an easy solution (if one exists) to the problem $P(f)$ in $O(n^4)$ time.

It should be noted that Lemma 2.4 is valid for NAESPI problems even if we drop the requirement that every pair of clauses has a non-empty intersection.

Next we show that if every pair of clauses in a given NAESPI problem $P(f)$ always intersects in more than one variable then $P(f)$ is trivially satisfiable.

**Lemma 2.5** NAESPI with cardinality of intersection $\geq 2$ is always solvable.

**Proof:** Let $C$ be the clause which does not properly contain any other clause (such a clause always exists). Let the cardinality of this clause be $m$. Pick any $m - 1$ elements from this clause and denote this set $S$. We claim that $S$ is a solution. Since $C$ intersects every other clause in at least two variables, $S$ has to have a variable common with every clause. No clause can have all its literals set to 1 either. This follows from the fact that every other clause contains a literal not in $C$ and that literal is assigned a value of 0.

As we will see in the next section, NAESPI instances which are not easily satisfiable give an alternate characterization of almost self-dual functions as an approximation to the class of self-dual functions proposed by Bioch and Ibaraki [5].
2.4 NAESPI and Almost Self-Dual functions

In this section we study the relationship between easily satisfiable NAESPI instances and almost self-dual functions [5]. In particular we give an alternative characterization of almost self-dual functions. We assume that the inputs (monotone boolean functions) are in disjunctive normal form. We begin with some more definitions.

Definition 2.5 (Dual-minor) \( f \) is dual-minor if \( f \leq f^d \).

Given \( w \), a minterm of \( f \), we represent by \( \bar{w} \) the product of all the variables which are not in \( w \) but are in \( f \). \( w^d \) is the dual of \( w \).

Definition 2.6 (Sub-dual) Subdual \( f^s \) of a function \( f \) is defined as \( \sum_{w \in f} w w^d \), where \( w \) is a minterm in \( f \).

Definition 2.7 (Almost dual-major) A function \( f \) is almost dual-major if \( f^s \leq f \).

A function \( f \) (in DNF) is satisfiable if there exists a vector \( x \in \{0, 1\}^n \) such that some term evaluates to 1. The set of variables set to 1 is referred to as the solution set \( S \). \( f \) is easily satisfiable if the solution set \( S \) is properly contained in some term in \( f \).

Definition 2.8 (Almost self-dual) A function \( f \) is called almost self-dual if \( f \) is almost dual-major and at the same time dual-minor.

By definition all self-dual functions are almost self-dual but the converse is not true. The following is one such example from the paper of Bioch and Ibaraki [5] attributed to Makino.

Example 2.1

\[
\begin{align*}
f &= (1 \land 2 \land 4) \lor (1 \land 2 \land 5) \lor (1 \land 3 \land 5) \\
    &\lor (1 \land 3 \land 6) \lor (1 \land 5 \land 6) \lor (2 \land 3 \land 5) \\
    &\lor (2 \land 3 \land 6) \lor (2 \land 5 \land 6) \lor (3 \land 4 \land 5)
\end{align*}
\]
$f^* =
(1 \land 2 \land 3 \land 4) \lor (1 \land 2 \land 3 \land 6) \lor
(1 \land 2 \land 4 \land 5) \lor (1 \land 2 \land 4 \land 6) \lor
(1 \land 2 \land 5 \land 6) \lor (1 \land 3 \land 4 \land 5) \lor
(1 \land 3 \land 4 \land 6) \lor (1 \land 3 \land 5 \land 6) \lor
(1 \land 4 \land 5 \land 6) \lor (2 \land 3 \land 4 \land 5) \lor
(2 \land 3 \land 4 \land 6) \lor (2 \land 3 \land 5 \land 6) \lor
(2 \land 4 \land 5 \land 6) \lor (3 \land 4 \land 5 \land 6)$

$f^d =
(1 \land 2 \land 3) \lor (1 \land 2 \land 4) \lor (1 \land 2 \land 5) \lor
(1 \land 3 \land 4) \lor (1 \land 3 \land 6) \lor (1 \land 5 \land 6) \lor
(2 \land 3 \land 4) \lor (2 \land 3 \land 5) \lor (2 \land 3 \land 6) \lor (2 \land 5 \land 6) \lor (3 \land 4 \land 5) \lor (4 \land 5 \land 6)$

The next theorem characterizes almost self-dual functions in terms of instances of NAE-SPI which admit easy solutions.

**Lemma 2.6** A monotone boolean function $f$ is almost self-dual $\iff f^d = P(f)$ does not have any easy solution.

*Proof:* $\Rightarrow$ Given an almost self-dual function $f$, we want to prove that $f^d$ is not easily satisfiable. As $f$ is self-dual, $f^* \leq f$, which implies $f^d \leq (f^*)^d$. Suppose that $f^d$ is easily satisfiable. This implies $(f^*)^d$ evaluates to 1 on the same vector $x$. Let $x$ be properly contained inside clause $C \in f^d$. But in $(f^*)^d$ we have $\bar{C}$ as a clause and as $(f^*)^d$ is in conjunctive normal form, $x$ is not a solution to $f$.

$\Leftarrow$ Given that $f^d$ does not have any easy solution, we want to show that $f^d \leq (f^*)^d$. Suppose that $f^d(x) = 1$ for some vector $x$. We want to show that $(f^*)^d(x) = 1$ on the same vector $x$. As the solution to $f^d$ is a strong solution, it intersects every clause in $(f^*)^d$. It cannot contain a clause in $(f^*)^d$, otherwise the solution would have been an easy solution. $f$ is dual-minor because $f$ has the intersection property. This follows from the fact that $\forall x \in \{0, 1\}^n$ $(f(x) = f(\bar{x})) \neq 1$, else we have two terms $T_i, T_j$ in $f$ such that $T_i \cap T_j = \phi$. $\blacksquare$

$\footnote{f \leq g \iff g^d \leq f^d.}$
Bioch and Ibaraki [5] also define an operator $\rho$ on monotone boolean functions which preserves self-duality. Given $f$ and a minterm $w$ of $f$, $\rho_w(f) = \bar{w} + w\bar{w} + (f \setminus w)$, where $A \setminus B$ denotes elements that are in the set $A$ but not in the set $B$.

An analogous operator for the NAESPI can be defined. Given a NAESPI problem $P(f)$, we define $\rho$ for a clause $C$ as $\rho_C(P(f)) = \bar{C} \land \{v \in C (\bar{C} \lor v)\} \land (P(f) \setminus C)$. Let us look at the following NAESPI defined by the projective plane of order 3.

Example 2.2

$$P(f) = ((1 \lor 2 \lor 3) \land (3 \lor 4 \lor 5) \land (1 \lor 5 \lor 6) \land (1 \lor 4 \lor 7) \land (2 \lor 5 \lor 7) \land (3 \lor 6 \lor 7) \land (2 \lor 4 \lor 6))$$

Then $\rho$ applied on clause $(1, 2, 3)$ is

$$\rho_{\{1, 2, 3\}}(P(f)) = ((4 \lor 5 \lor 6 \lor 7) \land (4 \lor 5 \lor 6 \lor 7 \lor 1) \land (4 \lor 5 \lor 6 \lor 7 \lor 2) \land (4 \lor 5 \lor 6 \lor 7 \lor 3) \land (3 \lor 4 \lor 5) \land (1 \lor 5 \lor 6) \land (1 \lor 4 \lor 7) \land (2 \lor 5 \lor 7) \land (3 \lor 6 \lor 7) \land$$
It should be noted that the application of this operator does not return an instance in minimal form, in that there might be clauses which contain other clauses in the problem.

It has been shown in [5] that if $f$ is a self-dual function then any number of applications of the $\rho$ operator preserves self duality. If $f$ is not self-dual then it can happen that after repeated applications of the $\rho$ operator, the instance can become easily satisfiable. This characterization leads to the following algorithm for detecting self-duality (proposed in [5]). The input is a monotone boolean function of $m$ variables. We are interested in determining if $f$ is self-dual. Let $g$ be a known self-dual function over $m$ variables. For a function $f$, the set of all true (false) vectors is denoted by $T(f)(F(f))$ respectively. The minimal set of true (false) vectors is denoted by $\text{min}(T(f))(\text{min}(F(f)))$ respectively. The algorithm is as follows:

1. If $f = g$ return yes
2. If either $f < g$ or $f$ has an easy solution then output no and halt.
3. Take some $w \in \text{min}(T(f)) \cap F(g)$ and let $f = \rho_w(f)$ and return to Step 1.

It has been argued in [5] that the above mentioned algorithm is not polynomial. The complexity of this procedure is a function of $|T(f) \setminus T(g)|$.

It should be noted that by repeated applications of the $\rho_w$ operator, we might obtain a problem which contains strictly fewer variables than in the original problem. This observation suggests the following variation on the algorithm proposed in [5].

We are given an instance $P(f)$ of NAESPI with $m$ variables. We want to determine if the input is unsatisfiable, using the recursive procedure outlined below.

1. Let $C$ be the smallest-sized clause. If $P(f) = \rho_C(P(f))$ contains fewer number of variables than the original problem, then solve $P(f) = \rho_C(P(f))$ recursively.
2. If the number of variables is 3 and if $P(f)$ is isomorphic to $f_3$ (defined below) then return unsatisfiable else return satisfiable.

---

2 We use $n$ to denote the number of clauses in $P(f)$ hence the choice of $m$. 
This algorithm was tested on unsatisfiable instances of NAESPI generated according to the following definition.

1. \( f_3 = ((1 \lor 2) \land (1 \lor 3) \land (2 \lor 3)) \).

2. \( f_i \) for odd \( i \) contains all the clauses in \( f_{i-2} \) augmented once with variable \( i \) once and once with variable \( i - 1 \). The last clause is \( (i \lor i - 1) \).

An example of an unsatisfiable NAESPI with 5 variables generated according to the method outlined above is given below.

**Example 2.3**

\[
\begin{align*}
(4 \lor 5) & \land \\
(1 \lor 2 \lor 4) & \land \\
(1 \lor 3 \lor 4) & \land \\
(2 \lor 3 \lor 4) & \land \\
(1 \lor 2 \lor 5) & \land \\
(1 \lor 3 \lor 5) & \land \\
(2 \lor 3 \lor 5) & \\
\end{align*}
\]

We are interested in the number of steps needed to reduce the number of variables in the problem at which point it can be solved recursively. As the number of clauses is exponential in the number of variables, the maximum problem size we tried had 15 variables.

The first column in the following table is the number of variables in the problem and the second column is the number of \( \rho \) iterations needed to reduce the size of the problem by one variable.
We can also analyze a randomized variation of this algorithm theoretically. Let us consider the following variation, where in Step 2 of the algorithm instead of choosing the smallest-sized clause we choose a clause at random. The algorithm can be analyzed by using the graph defined in [5]. \( G = (V, E) \) is an undirected graph, with \( V \) as the set of all the self-dual functions over \( m \) variables. There is an edge \((P(f), P(f)')\) if \( \rho_C(P(f)) = P(f)' \) (this means that by applying \( \rho \) on \( P(f) \) we obtain \( P(f)' \)). The next lemma gives us a lower bound on the number of self-dual functions on \( m \) variables. In the variation of the algorithm we are interested in, the algorithm can now be thought of as a random walk on the graph \( G \). As \( G \) is connected [5], the expected number of steps needed to reach a vertex from any specified vertex is \( O(|V|^2) \) [40].

Let \( \mathcal{SD}(m) \) be the number of self-dual functions of \( m \) variables. The following bound on \( \mathcal{SD}(m) \) establishes that the number of vertices in the graph \( G \) under consideration is exponential in \( m \).

**Lemma 2.7** \( \mathcal{SD}(m) \geq 2^{\frac{m}{2}} \).

*Proof: \( \mathcal{SD}(m) \geq 2 \times \mathcal{SD}(m - 2) \) for odd \( m \). The first instance is obtained by the recursive construction shown above. The second one is obtained by applying the \( \rho(m - 1 \lor m)(P(f)) \) operator. The previous recurrence gives the bound mentioned above.*

Therefore, in the worst case, the \( \rho \) operator might have to be applied \( \mathcal{SD}(m) \) times until the number of variables can be decremented by 1. In fact, it is known that \( \mathcal{SD}(m) \geq 2^{\left(\frac{m-1}{2}\right)} \) [20]. This was brought to our attention by Ibaraki.
2.5 NP-Completeness

In this section we show that finding a particular type of solution (called strong solution) to NASEPI is NP-Complete. To attain this, we first show that finding strong solutions to NAESP is NP-Complete. Observe that this does imply NP-Completeness of finding a solution to NAESPI. An explanation of this fact is given at the end of this section.

**Definition 2.9 (NAESP)** The generalized version of NAESPI where the intersection property need not hold is called NAESP.

**Definition 2.10 (Strong Solution)** A solution $S$ to NAESPI is a strong solution if neither $S$ nor $\tilde{S}$ is an easy solution.

**Definition 2.11 (Strongly satisfiable)** An instance of NAESPI is strongly satisfiable if it admits a strong solution.

2.5.1 Finding strong solutions to NAESPI is NP-Complete

**Definition 2.12** An instance of NAESP is called irredundant if it has no clause which contains any other clause.

Henceforth, we will assume that the given NAESP problem is irredundant. We will reduce 3-NAES (Not All Equal Satisfiability with three literals) a well-known NP-Complete problem [21] to irredundant 3-NAESP, thereby establishing the NP-Completeness of irredundant NAESP.

**Lemma 2.8** Irredundant NAESP is NP-Complete.

**Proof:** Let $m$ be the number of clauses and $n$ the number of variables in 3-NAES. We replace each $\overline{x_i}$ with a new variable $x_{n+i}$ in all the clauses. For all $(x_i, \overline{x_i})$ we add the following four clauses $(x_i \lor x_{n+i} \lor a) \land (x_i \lor x_{n+i} \lor b) \land (x_i \lor x_{n+i} \lor c) \land (a \lor b \lor c)$. The NAESP problem so generated is irredundant.

$\Rightarrow$ If 3-NAES is satisfiable then the set of clauses generated is also satisfiable. The solution is obtained by setting $x_{n+i}$ to the same value as $\overline{x_i}$ and $a$ to 0 and $b, c$ to 1.
\section*{CHAPTER 2. \textit{NAESPI AND SELF-DUALITY}}

\begin{itemize}
\item[\textless ;] If the newly generated set of clauses is satisfiable then the solution to 3-\textit{NAES} is obtained by setting $x_i$ to the same value as $x_{n+i}$. This assignment clearly satisfies all the clauses. We have to show that a variable and its negation are not both 1 or 0 in the solution. If this were the case then the clause $(a \lor b \lor c)$ would not be satisfied. \hfill \blacksquare
\end{itemize}

\textbf{Lemma 2.9} \textit{Finding strong solutions to irredundant \textit{NAESP} is NP-Complete.}

\textit{Proof:} Let $I$ be the set of all the instances of irredundant \textit{NAESP}. $I$ can be divided into two sets: the first set admits only easy solutions and the remaining set admits strong solutions. If we can find strong solutions to \textit{NAESP} in polynomial time then in the same amount of time we can determine the satisfiability of irredundant \textit{NAESP}. Hence determining strong solutions is as hard as determining solutions to \textit{NAESP}. \hfill \blacksquare

\textbf{Lemma 2.10} \textit{Finding strong solutions to \textit{NAESPI} is NP-Complete.}

\textit{Proof:} We reduce \textit{NAESP} problems which admit only strong solutions to \textit{NAESPI}. Given an instance $P$ of \textit{NAESP}, let $A$ be the set of clauses which contain some variable denoted ‘1’. Let $B$ be the remaining set of clauses. If there are $n$ variables in $P$ then the complement of a clause is defined as the set complement of the variables in the clause. We define $P(f)$ as $A \cup B'$ where every clause in $B$ is complemented in $B'$. $P(f)$ clearly obeys the intersection property as all the clauses contain variable 1.

Next we need to prove that $S$ is a strong solution to $P \iff S$ is also a strong solution to $P(f)$.

\begin{itemize}
\item [$\Rightarrow$] Let $S$ be a strong solution to $P$. Clearly $S$ intersects every clause in $P(f)$. There are two cases:

\begin{enumerate}
\item Let $q$ be a clause in $P(f)$ and $S \subset q$. Clearly $q \notin A$ since $S$ is a strong solution. Thus $q \in B'$. This implies $\bar{q} \in B$, which contradicts the fact that $S$ is a solution to $P$, as $S$ does not intersect with $\bar{q}$.

\item $q \in P(f) \subset S$. Again $q \in B'$. Since $S$ is solution $\bar{S} \subset \bar{q}$. This violates the fact that $S$ was a solution to $P$.
\end{enumerate}

\item [$\Leftarrow$]
\end{itemize}
1. Let $q$ be a clause in $P$ and $S \subseteq q$. Clearly $q \notin A$ since $S$ is a strong solution. Thus $q \in B'$. This implies $\bar{q} \in B'$, which contradicts with the fact that $S$ is a solution to $P(f)$, as $S$ does not intersect with $\bar{q}$.

2. $q \in P \subseteq S$. Again $q \in B'$. Since $S$ is solution $\bar{S} \subseteq q$. This violates the fact that $S$ was a solution to $P(f)$.

It should be noted that NP-Completeness of finding strong solutions to N-4ESPI does not imply that finding a solution to N-4ESPI is NP-Complete, as in our construction every instance of N-4ESPI has an easy solution: the variable denoted '1'.

2.6 The constant case

In this section we study $k$-N-4ESPI in which every clause has at most $k$ variables for some constant $k$. First we show that 3-N-4ESPI can be solved in polynomial time. A straightforward generalization of the 3-N-4ESPI result to any constant $k$ does not work but we present an algorithm which solves the $k$-N-4ESPI problem with $n$ clauses for any constant $k$ in $O((nk)^{k+1})$ time. We then improve on the complexity of the previous algorithm to achieve an $O((2^k)^{nk})$ algorithm. The $O((2^k)^{nk})$ algorithm for solving $k$-N-4ESPI is obtained by modifying the $O((nk)^{k+1})$ algorithm. For the sake of exposition we present both the algorithms in this section even though $O((2^k)^{nk})$ algorithm clearly has a better running time than the $O((nk)^{k+1})$ algorithm.

2.6.1 3-N-4ESPI

In this section we demonstrate a polynomial-time algorithm for the 3-N-4ESPI problem. Each clause in 3-N-4ESPI problem contains at most 3 variables, therefore at least one of the six assignments of 0/1's to the variables in every clause should be extendible to a solution, if the problem were satisfiable. Our approach is to show that for each of these assignments the subproblem is a 2-SAT problem. As 2-SAT can be solved in time linear in the number of clauses in the problem, the algorithm for 3-N-4ESPI terminates in polynomial-time.

For the purpose of exposition, consider the following 3-N-4ESPI:

$$(1 \lor 2 \lor 3) \land (3 \lor 4 \lor 5) \land (1 \lor 5 \lor 6) \land (1 \lor 4 \lor 7) \land (2 \lor 5 \lor 7) \land (3 \lor 6 \lor 7)$$
CHAPTER 2. NAESPI AND SELF-DUALITY

Let us say that we are going to try out all the 6 assignments of 1's and 0's to the variables in the first clause \((1 \lor 2 \lor 3)\). Suppose that the first assignment we try is variables 1 and 2 set to 0 and variable 3 set to 1. With this assignment we get two sets of clauses, \(P\) and \(N\), where set \(P\) comprises only those clauses which intersect the first clause only in the variables set to 1, and set \(N\) comprises clauses which intersect the first clause only in the variables set to 0. Clauses containing both types of variables can be ignored. For the above example,

\[
P = (3 \lor 4 \lor 5) \land (3 \lor 6 \lor 7)
\]

and

\[
N = (1 \lor 5 \lor 6) \land (1 \lor 4 \lor 7) \land (2 \lor 5 \lor 7)
\]

No clause in the set \(P\) can have all the variables set to 1. This is equivalent to saying that the following clauses are satisfiable:

\[
(\overline{4} \lor \overline{5}) \land (\overline{6} \lor \overline{7})
\]

Also, every clause in the set \(N\) has to contain at least one variable set to 1. This is equivalent to saying that the set

\[
(5 \lor 6) \land (4 \lor 7) \land (5 \lor 7)
\]

is satisfiable. Combining the previous two sets of clauses which have to be satisfied, we can say that the assignment of 0/1's to the variables 1,2,3 can be extended to a solution if the set \(PN\) of clauses is satisfiable:

\[
PN = (\overline{4} \lor \overline{5}) \land (\overline{6} \lor \overline{7}) \land (5 \lor 6) \land (4 \lor 7) \land (5 \lor 7)
\]

Since \(PN\) is a 2-SAT problem, we can solve it in polynomial time. Formally we can state the result as the following theorem:

**Theorem 2.1** 3-NAESPI is polynomially solvable.

*Proof:* Let the variables in a clause \(C\) be assigned 0/1 values. If a literal is assigned value 1 we say that it is in the solution. We now generate two sets of clauses, \(P\) and \(N\). \(P\) comprises those clauses which intersect \(C\) only in the variables which are assigned value 1. Similarly
CHAPTER 2. NAESPI AND SELF-DUALITY

\( N \) comprises those clauses which intersect \( C \) only in the variables which are assigned value zero. Clauses which intersect \( C \) in variables which have values 0 and 1 are discarded since they are already satisfied. Next we remove from \( P \) and \( N \) all the variables that have been assigned a value. Now all the clauses in \( N \) have to be satisfied and no clause in \( P \) can have all the variables set to 1. Clauses in the set \( P \) can be written down as satisfiability conditions by complementing each literal in the clause. Let this set be denoted by \( P' \). \( P' \cup N \) is 2-SAT which can be solved in polynomial time. It should be noted that because of the intersection property we have covered all the clauses in the original problem. If the new problem is satisfiable we have a solution; otherwise, we change our assignment of values to the variables in \( C \). Since there are at most six different assignments to be tried, the overall time taken is linear in the number of clauses.

2.6.2 \( k \)-NAESPI

In this section we describe a \( O((nk)^{k+1}) \) algorithm for solving the \( k \)-NAESPI problem. For a given clause in the \( k \)-NAESPI problem, there are at most \( k \) assignments of 0/1's to the variables which set exactly one of the variables in the clause to 1 and the rest of the variables to 0. We call such an assignment a 10\(^*\) assignment. The algorithm operates in stages. For each subproblem \( j \) in stage \( i \) let \( U_{i,j} \) be the set of clauses which do not contain any variable set to 1. the algorithm tries out all the \( k \) possible assignments of the type 10\(^*\) for all the clauses in \( U_{i,j} \). If at most \( k \) stages are needed by the algorithm then the total amount of effort needed is \( O((nk)^{k+1}) \).

Let us demonstrate the algorithm by means of the following example in which the clauses have been ordered from left to right:

\[
(1 \lor 2 \lor 3) \land (1 \lor 4 \lor 5) \land (2 \lor 4 \lor 6) \land (3 \lor 5 \lor 6)
\]

\( U_{1,1} \) initially is the set of all the clauses In Stage 1, the algorithm has to try out three possible assignments each for the four clauses. By \( x \rightarrow a \) we denote the fact that variable \( x \) has been assigned the value \( a \). The 12 assignments which the algorithm tries out in Stage 1 are:

1. \( 1 \rightarrow 1, 2 \rightarrow 0, 3 \rightarrow 0 \)
2. 1 − 0, 2 − 1, 3 − 0
3. 1 − 0, 2 − 0, 3 − 1
4. 1 − 1, 4 − 0, 5 − 0
5. 1 − 0, 4 − 1, 5 − 0
6. 1 − 0, 4 − 0, 5 − 1
7. 2 − 1, 4 − 0, 6 − 0
8. 2 − 0, 4 − 1, 6 − 0
9. 2 − 0, 4 − 0, 6 − 1
10. 3 − 1, 5 − 0, 6 − 0
11. 3 − 0, 5 − 1, 6 − 0
12. 3 − 0, 5 − 0, 6 − 1

This means that in Stage 2 there are $n \times k$ subproblem(s) containing at most $n - 1$ clauses of size at most $k$. Suppose we are looking at the subproblem obtained by the first choice of values for the variables. Then $U_{2,1}$ which is the set of clauses which do not contain any variable set to 1 is

$$U_{2,1} = (2 \lor 4 \lor 6) \land (3 \lor 5 \lor 6)$$

similarly,

$$U_{2,2} = (1 \lor 4 \lor 5) \land (3 \lor 5 \lor 6)$$

$$U_{2,3} = (1 \lor 4 \lor 5) \land (2 \lor 4 \lor 6)$$

$$U_{2,4} = U_{2,1}$$

$$U_{2,5} = (1 \lor 2 \lor 3) \land (3 \lor 5 \lor 6)$$

$$U_{2,6} = (1 \lor 2 \lor 3) \land (2 \lor 4 \lor 6)$$

$$U_{2,7} = U_{2,2}$$
The algorithm then recurses on each of the subproblems generated. We later show that the number of distinct subproblems generated from a subproblem in the previous stage (any Stage i) is at most \( nk \). Assuming that at most \( k \) stages are needed (proof of which is presented later), the total number of subproblems generated is \( \leq O((nk)^k) \). As it takes \( O(nk) \) time to verify a solution the total time taken is \( O((nk)^k nk) \). We will give a formal proof of this in subsequent paragraphs.

To prove the correctness of the algorithm we need the concept of minimal solutions.

**Definition 2.13 (Minimal)** A solution \( S \) to \( k \)-NAESPI is minimal if no proper subset of \( S \) is a solution.

Suppose that the given \( k \)-NAESPI is solvable. Let \( S \) be a minimal solution. Then we have a clause \( C \) which contains at most one element from \( S \). Suppose this were not true, then every clause contains at least two variables from \( S \). Remove from \( S \) any element \( s \in S \). \( S - \{s\} \) is still a solution to the problem as every clause contains at least one 1 now. Clearly this violates the minimality of \( S \). We can use the same argument for any stage \( i \).

**Lemma 2.11** If the partial assignment \( A \) can be extended to a complete solution \( A' \) then there exists a clause in \( U \) which contains at most one element from \( A' \) provided \( A' \) is minimal.

**Proof:** Assume that we are at Stage \( i \). At this stage we have set some variables to 0/1. Let us denote this partial assignment of variables set to 1 as \( A \). Let \( U \) be the set of clauses which do not contain any of the variables set to 1. Let \( W \) be the set of clauses which have been removed as they contain some variable set to 1. \( A \) is the set of variables which are set to 1. Let \( B \) be the set of variables occurring in the clause set \( U \). It is clear that \( A \cap B = \emptyset \). This implies that setting any variable in \( B \) to 0 does not unsatisfy any clause which was
satisfied in the set \( W \). To obtain a contradiction assume that every clause in \( U \) contains at least two variables from the set \( A' \). We can set any variable in \( A' \) to 0 without unsatisfying any of the previously satisfied clauses. This violates the fact that \( A' \) was minimal.

Next we need to show that the satisfiability of any subproblem at Stage \( k+1 \) is easy to determine. We will achieve this by arguing that if we have \( k \) distinct clauses each of cardinality at most 2 in a \( k \)-NAESPI problem then we can determine the satisfiability easily. Furthermore, after the algorithm has made \( k \) choices (or is in Stage \( k+1 \)) the resulting problem is equivalent to a problem which has \( k \) clauses of cardinality 2.

**Lemma 2.12** Let \( P(f) \) be a \( k \)-NAESPI problem which contains \( k \) distinct clauses of size 2. Satisfiability of \( P(f) \) can be determined in polynomial time.

**Proof:** Without loss of generality assume that the first \( k \) clauses are

\[
(a_1 \lor b) \land (a_2 \lor b) \land \ldots \land (a_k \lor b)
\]

Suppose that there exists a clause \( q \) that does not contain \( b \). Then \( a_i \in q, 1 \leq i \leq k \), for the intersection property to be obeyed. If such a \( q \) exists then \( P(f) \) is unsatisfiable, else \( P(f) \) is satisfiable. The solution can be obtained by assigning \( a_1, a_2, \ldots, a_k \) value 1 and all the other variables value 0.

The algorithm moves from Stage \( i \) to Stage \( i+1 \) by picking a clause and setting some variable in it to 1 and every other variable to 0. The variable set to 1 is different from any of the variables which were set to 1 in the past. This follows from the fact that at each stage the algorithm only considers clauses which do not have any variable set to 1 in them. Suppose that our algorithm is in Stage \( k+1 \). This means that there are at least \( k \) clauses which have exactly one variable in them set to 1 and all the other variables set to 0. Also, the \( k \) variables which have been set to 1 are all distinct. We next define the concept of *contraction* and describe some properties of contractions. We will use contraction to show
that the problem after \( k \) stages has an equivalent formulation in which there are \( k \) distinct clauses of cardinality 2.

Let \( A \) be a subset of variables in the \( k \)-NAESPI problem.

**Definition 2.14 (Contraction)** A contraction of a subset \( A \) of variables in a \( k \)-NAESPI problem \( P(f) \) to \( a \) is a new problem \( P(f)' \) in which every occurrence of any member of \( A \) is replaced by the new variable \( a \). Contraction is denoted by \( A \rightarrow a \).

For example let us look at the following 3-NAESPI.

\[
(1 \lor 2 \lor 3) \land (1 \lor 4 \lor 5) \land (2 \lor 4 \lor 6) \land (3 \lor 5 \lor 6)
\]

We want to contract \( \{1, 2, 4\} \) to \( a \), i.e. \( \{1, 2, 4\} \rightarrow a \). This means that we replace every occurrence of either 1, 2 or 4 with the new variable \( a \). The resulting problem is:

\[
(a \lor 3) \land (a \lor 5) \land (a \lor 6) \land (3 \lor 5 \lor 6)
\]

For a \( k \)-NAESPI problem \( P(f) \) let \( P(f)' \) be the problem obtained by the contraction \( A \rightarrow a \) for some subset \( A \) of the variables in \( P(f) \). The next property of contractions follows from the definition.

**Lemma 2.13** \( P(f)' \) is satisfiable \( \iff \) \( P(f) \) has a solution \( S \) which contains all the variables in \( A \).

Contraction tells us that some subset \( A \) of the variables in the problem are forced to have the same value. Let us get back to our problem after our algorithm has executed \( k \) stages.

**Lemma 2.14** After the algorithm has made \( k \) choices (is in Stage \( k+1 \)) there exists a contraction such that the resulting problem \( P(f)' \) has \( k \) clauses of cardinality 2.

We have \( k \) clauses of the type

\[
(a_1 \lor A_1) \land
\]
\[(a_2 \lor A_2) \land \\
\ldots \land \\
(a_k \lor A_k)\]

\[a_1, a_2, \ldots, a_k \text{ are distinct variables and } A_1, A_2, \ldots, A_k \text{ are subsets of the variables. As } A_1 = A_2 = \ldots = A_k = 0 \text{ we can contract } (A_1 \cup A_2 \cup \ldots \cup A_k) \text{ to a new variable } b. \] This means that the \(k\) clauses look like

\[(a_1 \lor b) \land \\
(a_2 \lor b) \land \\
\ldots \land \\
(a_k \lor b)\]

and are of cardinality 2. \[\blacksquare\]

Armed with the last two results we can show that the algorithm terminates in \(O((nk)^{k+1})\) time.

**Theorem 2.2** The algorithm stated above terminates in polynomial time.

**Proof:** By Lemma 2.14 the algorithm needs at most \(k\) stages. In each stage the algorithm has to try at most \(n \times k\) assignments. Therefore the total number of subproblems generated is given by the following recurrence:

\[T(k) = n \times k \times T(k - 1)\]

which evaluates to \(O(n \times k)^k\). As it takes \(O(nk)\) time to verify a solution the total time taken is \(O((n \times k)^{k+1})\). \[\blacksquare\]

In this section we have shown that \(k\)-NAESPI can be solved in \(O((nk)^{k+1})\) time.
2.7 Linear Time algorithm for solving $k$-NAESPI

The algorithm is again recursive but instead of trying out every possible 10* assignment for every clause, it tries out all the $2^k - 2$ contractions for each of some $k$ clauses. The algorithm begins by choosing a clause of length greater than 2 and a contraction for it. It then removes all the clauses which are trivially satisfied (clauses which contain both the contracted variables). Suppose that we are in Stage $l+1$. The clauses which are of cardinality 2 are of the form shown below (Lemma 2.12). This is under the assumption that the contraction we made is extendible to a solution.

$$
(a_1 \lor b) \land \\
(a_2 \lor b) \land \\
(a_3 \lor b) \land \\
(a_4 \lor b) \land \\
(a_i \lor b)
$$

Without loss of generality, we assume that there is a clause which does not contain any of the variables from the set $\{a_1, \ldots, a_i\}$, else we have a solution to the problem. This follows from the fact that each clause contains at least one of the variables from $\{a_1, \ldots, a_i\}$. Setting all the variables in $\{a_1, \ldots, a_i\}$ to 1 and the rest of the variables to 0, results in a solution to the given instance. Let $C$ be such a clause. For Stage $l+1$ we try out all the possible $2^k-2$ contractions for Clause $C$. We need to argue that any contraction of $C$ gives us a subproblem with $l+1$ distinct variables $a_1, a_2, \ldots, a_{l+1}$. Let $A$ be the set of variables in $C$ which are set to the same value and $B$ the set of remaining variables in $C$ (which are set to a value different from the value to which the variables in $A$ are set). If $b \not\in A$ then there exists a variable in $B$ which is different from any of the $a_1, a_2, \ldots, a_{l}$. This is due to the fact that $C$ does not contain any of the variables $a_1, a_2, \ldots, a_{l}$. Hence the clause obtained after the contraction is distinct. The case when $b \in A$ is symmetrical.

Formally the algorithm is stated below:

**Algorithm**

1. $S$ is the set of distinct variables which belong to some clauses of size 2 and are forced to have the same value ($S = \{a_1, a_2, \ldots, a_l\}$ in the previous example). Initially $S = \Phi$. 
2. Find a clause \( C \) such that \( C \) does not contain any variable in \( S \). If no such clause exists then \( S \) intersects with all the clauses and we are done.

3. For each contraction (out of the \( 2^k - 2 \) possible ones), update \( S \), remove all the clauses which are trivially satisfied and goto Step 2.

Let us consider the projective plane example again.

Example 2.4

\[(1 \lor 2 \lor 3) \land (3 \lor 4 \lor 5) \land (1 \lor 5 \lor 6) \land (1 \lor 4 \lor 7) \land (2 \lor 5 \lor 7) \land (3 \lor 6 \lor 7) \land (2 \lor 4 \lor 6)\]

Consider the first clause and a contraction in which \( \{1, 2\} \) gets the same value and \( 3 \) gets a value different from \( 1 \) and \( 2 \). Since \( \{1, 2\} \) get the same value we can replace them with a new variable \( a \). Hence, the modified problem is:

\[(a \lor 3) \land (3 \lor 4 \lor 5) \land (a \lor 5 \lor 6) \land (a \lor 4 \lor 7) \land (a \lor 5 \lor 7) \land (3 \lor 6 \lor 7) \land (a \lor 4 \lor 6)\]

and \( S = \{a\} \). Let \((3 \lor 4 \lor 5)\) be the clause \( C \) (which does not contain \( a \)) for which we are going to try out all the possible contractions next. Possible contractions for \( C \) are \( \{\{3, 4\}, \{3, 5\}, \{4, 5\}\} \). Let \( \{3, 4\} \) be contracted to variable \( b \). Then the subproblem obtained is:

\[(a \lor b) \land (b \lor 5) \land (a \lor 5 \lor 6) \land (a \lor b \lor 7) \land (a \lor 5 \lor 7) \land (b \lor 6 \lor 7) \land (a \lor b \lor 6)\]

\( S \) now is updated to \( S \cup \{5\} = \{a, 5\} \). Also, the problem is not in minimal form as we have clauses which contain the clause \( (a \lor b) \). The minimal subproblem is:

\[(a \lor b) \land (b \lor 5) \land (a \lor 5 \lor 6) \land (a \lor 5 \lor 7) \land (b \lor 6 \lor 7)\]

and so on.

The algorithm solves the subproblem recursively. If the subproblem is unsatisfiable then we try out the next contraction for the first clause. If all the contractions have been tried for the first clause then we return unsatisfiable.

**Theorem 2.3** The modified algorithm terminates in \( O((2^k)^k \times n \times k) \) time.
Proof: After \( k \) recursive calls we can use Lemma 2.14 to determine the satisfiability of the instance, as all the contracted clauses (of size 2) are distinct. Therefore the number of the times Lemma 2.14 is invoked is given by the following recurrence:

\[
f(k) = 2^k f(k - 1)
\]

As it takes \( O(nk) \) time to determine the satisfiability in the invocation of Lemma 2.14, the running time of the algorithm is \( O((2^k)^k \times n \times k) \) which is linear in \( n \).

Consider the following restriction to the NAESPI in which every clause is of size at most \( \log n \). The two algorithms mentioned in this chapter can be used to solve this restricted version of the NAESPI in \( O(n^{\log n}) \) time. The following question is open to the best of our knowledge:

**Open Problem 1** Let \( C \) be the class of NAESPI such that all the clauses are of size at most \( \log n \). Does there exist an \( O(n^{\log \log n}) \) for this problem?

### 2.8 Note

Recently a result of Domingo [8] was brought to our attention by Kaz Makino, which answers a restricted version of the previous question. Domingo showed that \( O(\sqrt{\log n}) \)-NAESPI can be solved in polynomial time. In fact, Theorem 2.3 also implies a polynomial time algorithm for solving \( O(\sqrt{\log n}) \)-NAESPI.

### 2.9 Conclusion

In this chapter we established the equivalence of the complement of the NAESPI and the self-duality problem. We showed that easy solutions to the NAESPI can be computed in polynomial time, furthermore, computing strong solutions to NAESPI is NP-Complete. We gave a characterization of almost self-dual functions in terms of NAESPI which do not admit easy solutions. We exhibited an \( O((nk)^{k+1}) \) algorithm for solving the \( k \)-NAESPI problem for some constant \( k \). Finally, we provide a linear time algorithm for solving \( k \)-NAESPI, with a running time of \( O((2^k)^{nk}) \). This improvement resulted in a polynomial time algorithm.
for solving \((\sqrt{\log n})\cdot{\text{NAESPI}}\).
Chapter 3

The bounded case

3.1 Introduction

In this chapter we describe polynomial time algorithms for the $k$-NAESPI and the NAESPI problem when every pair of clauses intersect in at most $c$ variables. It should be noted that we treat $k$ and $c$ as constants in this chapter.

Definition 3.1 (c-bounded NAESPI) A $(k)$-NAESPI is $c$-bounded if every two clauses intersect in less than $c+1$ variables.

As pointed out in Chapter 1 $c$-bounded $k$-NAESPI is of interest because this subclass of NAESPI arises naturally in designing coteries used to achieve mutual exclusion in distributed system with the minimum number of messages.

For $c$-bounded $k$-NAESPI we show that there exists an algorithm which can determine the satisfiability of the input instance in $O(n^{c+1}k)$ time. We show an upper bound of $k^{c+1}$ on the number of clauses $(n)$ for $c$-bounded $k$-NAESPI which do not contain any solution of size strictly less than $c$. In this case the algorithm shown in Chapter 2 for solving $k$-NAESPI terminates in $O(k^{(c+1)k}n)$ time for $c$-bounded $k$-NAESPI. If there exists a solution of size at most $c$ then we try out all the subsets of size $c$. As there are $O(n^c)$ subsets of size $c$ and verifying the solution takes $O(nk)$ time, the total running time for this case is $O(n^{c+1}k))$. Since $O(n^{c+1}k))$ dominates $O(k^{(c+1)k}n)$, $c$-bounded $k$-NAESPI can be solved in $O(n^{c+1}k))$ time.
CHAPTER 3. THE BOUNDED CASE

For the $c$-bounded NAESPI we give an $O(n^{2c+2})$ algorithm for solving the problem. It should be noted that $c$-bounded $k$-NAESPI is a subclass of $c$-bounded NAESPI hence the latter results is weaker. Also, the techniques used in obtaining the respective results have no similarity whatsoever.

Sections 3.2 and 3.3 describe the results for the $c$-bounded $k$-NAESPI and $c$-bounded NAESPI problems respectively.

### 3.2 $c$-bounded $k$-NAESPI

In this section we show that for a $c$-bounded $k$-NAESPI, the number of clauses $n \leq k^{c+1}$. The main tool used in obtaining the results is an auxiliary graph which is defined below.

**Definition 3.2 (Auxiliary Graph)** An auxiliary graph is an edge labeled clique graph whose vertices are the clauses and the labels on edge $(i, j)$ are the variables which are common to clauses $i$ and $j$.

Figure 3.2 shows the auxiliary graph for the NAESPI problem derived from the projective plane coterie. The NAESPI instance comprises the following seven clauses:

\[
\begin{align*}
(1 \lor 2 \lor 3) & \land \\
(3 \lor 4 \lor 5) & \land \\
(1 \lor 5 \lor 6) & \land \\
(1 \lor 4 \lor 7) & \land \\
(2 \lor 5 \lor 7) & \land \\
(3 \lor 6 \lor 7) & \land \\
(2 \lor 4 \lor 6) & \land
\end{align*}
\]

The graph contains seven vertices, one for each clause and the edge $(u, v)$ contains labels corresponding to the variables which are common to both clauses $u$ and $v$. It should be noted that for a $1$-bounded $k$-NAESPI, every edge in the auxiliary graph contains exactly one label.
Definition 3.3 (c-solvable) A \( k \)-NAESPI is c-solvable if there exists a solution \( S \) such that \( |S| \leq c \).

For ease of exposition we describe the special cases of 1-bounded and 2-bounded \( k \)-NAESPI problems in the next subsections. A straightforward generalization of the results to the \( c \)-bounded \( k \)-NAESPI is presented later.

3.2.1 1-bounded \( k \)-NAESPI

Lemma 3.1 For 1-bounded \( k \)-NAESPI (which is not 1-solvable) the number of clauses \( n \leq k^2 \).
Proof: Let $G$ be the auxiliary graph and $u$ be the vertex corresponding to the clause $C$ with maximum cardinality. For any variable $x_i \in C$, if $x_i$ belongs to $l(x_i)$ clauses then there is a clique $K$ of size $l(x_i)$ such that every edge in the clique has label $x_i$.

Let $v$ be a vertex different from $u$ and any other vertex in $K$. Without loss of generality, we can assume that $v$ does not contain $x_i$ as a label, otherwise every clause contains variable $x_i$ (rendering the instance 1-solvable). Let $L$ be all the edges from $v$ onto $u$ and vertices in $K$. No two edges in $L$ have the same label. To derive a contradiction assume that two edges, incident on $u_1, u_2$, where both $u_1$ and $u_2$ are in $K$ have the same label $l$. This implies that the clauses corresponding to vertices $u_1$ and $u_2$ have variables $l$ and $x_i$ in common. This fact is illustrated in Figure 3.2.1 where the dotted ellipse is the clique $K$. 

Figure 3.2: 1-bounded NAESPI
This violates the fact that the given input is \( l \)-bounded.

From the argument presented above, we can deduce that

\[
l(x_i) \leq k \quad (1)
\]

Otherwise, the vertex \( v \) has more than \( k \) labels or equivalently we have a clause with cardinality bigger than \( k \).

We also know that,

\[
n \leq \sum_{x_i \in C} l(x_i) \quad (2)
\]

Equation (2) is a consequence of the intersection property. By Equation 1, we infer that

\[
\sum_{x_i \in C} l(x_i) \leq k^2
\]

Hence,

\[
n \leq k^2
\]

\[\Box\]

**3.2.2 2-bounded \( k \)-NAESPI**

In this section we take a closer look at 2-bounded \( k \)-NAESPI.

**Lemma 3.2** For 2-bounded \( k \)-NAESPI (which is not 2-solvable) the number of clauses \( n \leq k^3 \).

**Proof:** Let \( G \) be the auxiliary graph. For any two variables \( x_i \) and \( x_j \), let \( K_i \) and \( K_j \) be the cliques which contain labels \( x_i \) and \( x_j \). Let \( V_{i,j} = K_i \cap K_j \) be the set of vertices which are in \( K_i \) as well as \( K_j \). We claim:

\[
|V_{i,j}| \leq k
\]
Let $u$ be a vertex which is not in $K_i \cup K_j$. Such a vertex should exist otherwise the given input is 2-solvable. No two edges from $u$ which are incident on two vertices in $K_i \cap K_j$ can have the same label, else we have an edge which has 3 labels on it. As $|u| \leq k$, we get $|V_{ij}| \leq k$.

Now we bound the size of $K_i$. Let $v$ be a vertex which does not belong to $K_i$ ($v$ exists because the input is not 1-solvable). Every edge from $v$ onto $K_i$ has a label different from $x_i$ (as $K_i$ is maximal). Let $L$ be the set of labels on the edges incident from $v$ onto $K_i$. Each label $l \in L$ can occur at most $k$ times or we would have $|V_{i,l}| > k$. As $|L| \leq k$, maximum number of vertices in $K_i$ is $\leq k^2$.

From the previous paragraph, it follows that every variable $x$ belongs to at most $k^2$ clauses, i.e., $l(x_i) \leq k^2$. Let $C$ be a clause. We also know that,

$$n \leq \sum_{x_i \in C} l(x_i)$$

Hence,

$$n \leq k^3$$

3.2.3 \textit{c-bounded k-NAESPI}

In this section we study the $k$-NAESPI problem which is $c$-bounded. In particular, we show a generalization of the claims for the 1-bounded and 2-bounded cases.

\textbf{Theorem 3.1} For $c$-bounded $k$-NAESPI (which is not $c$-solvable) the number of clauses $n \leq k^{c+1}$.

\textit{Proof:} Let $G$ be the auxiliary graph. For any $c$ variables $x_1, \ldots, x_c$, let $K_1, \ldots, K_c$ be the corresponding cliques which contain labels $x_1, \ldots, x_c$. Let $V_{1,c} = \cap_{i \in \{1, \ldots, c\}} K_i$ be the set of vertices which are in cliques $K_1$ through $K_c$. We claim:

$$|V_{1,c}| \leq k$$
CHAPTER 3. THE BOUNDED CASE

Let \( u \) be a vertex which is not in \( K_1, \ldots, K_c \). Such a vertex should exist otherwise the given input is \( c \)-solvable. No two edges from \( u \) which are incident on any two vertices in \( V_{1,c} \) can have the same label, else we have an edge which has \( k + 1 \) labels on it. As \( |u| \leq k \), we get \( |V_{1,c}| \leq k \).

Now we bound the size of \( \cap_{i \in \{1,c-1\}} K_i \). Let \( v \) be a vertex which does not belong to \( \cap_{i \in \{1,c-1\}} K_i \) (\( v \) exists because the input is not \((c-1)\)-solvable). Every edge from \( v \) onto \( \cap_{i \in \{1,c-1\}} K_i \) has a label different from \( x_1, \ldots, x_{c-1} \) (as \( \cap_{i \in \{1,c-1\}} K_i \) is maximal). Let \( L \) be the set of labels on the edges incident from \( v \) onto \( \cap_{i \in \{1,c-1\}} K_i \). Each label \( l \in L \) can occur at most \( k \) times or we would have \( |V_{1,c}| > k \).

Using the argument presented above, if \( l \leq c \):

\[
|\cap_{i \in \{1,l\}} K_i| \leq k^{c-l+1}
\]

Now we are ready to bound the size of individual cliques \( K_i \). Let \( u \) be the vertex not in \( K_i \) (such a vertex exists because the input is not \( J \)-solvable). \( L \) is the set of labels on edges incident from \( u \) onto \( K_i \). We know \( |L| \leq k \) and \( |K_i \cap K_j| \leq k^{c-1} \) for any label \( x_j \). The maximum number of vertices in \( K_i \) is \( \leq k^c \) (i.e. \( l(x_i) \leq k^c \)).

We also know that,

\[
n \leq \sum_{x_i \in C} l(x_i)
\]

Hence.

\[
n \leq k^{c+1}
\]

In this section we have established that for instances of \( k \)-\( NAESPI \) which are \( c \)-bounded the number of clauses is \( n \leq k^{c+1} \). Next we describe an \( O(n^{2c+2}) \) algorithm for \( c \)-bounded \( NAESPI \).

3.3 \( c \)-bounded \( NAESPI \)

Definition 3.4 (\( c \)-bounded \( NAESPI \)) An instance of \( NAESPI \) is called \( c \)-bounded if every pair of clauses intersect in at most \( c \) variables for some constant \( c \).
CHAPTER 3. THE BOUNDED CASE

In this section we will show that c-bounded NAESPI can be solved in $O(n^{2c+2})$ time. For ease of exposition we will describe the 1-bounded and the 2-bounded NAESPI cases in subsections 1 and 2. The general result is presented in Subsection 3. For a set of variables $V$ an assignment of boolean values to the variables is called a 1*0 assignment if all the variables in $V$ except one are set to 1 and the remaining variable set to 0. If all but one variable are set to 0 then the assignment is called a 10* assignment.

We will be using the following definitions in the subsequent subsections. A solution $S$ to a given NAESPI is a subset of variables such that $V$ intersects each clause in the input but does not contain any clause in the input. A solution $S$ is called minimal if no proper subset of $S$ is a solution. If $V$ is the set of variables in the input instance then at times we refer to $S$ as the set of variables which can be set to 1 and $V \setminus S$ is the set of variables which can be set to 0. In the same vein as the presentation in the previous section, we describe the results for 1-bounded NAESPI and 2-bounded NAESPI first. The last subsection contains the result in the general form.

3.3.1 1-bounded NAESPI

In this section we will show that 1-bounded NAESPI, is solvable in $O(n^4)$ time. We observe that if a given NAESPI is solvable then there exists a clause such that it contains exactly one variable from the solution (this follows from the definition of the minimal solution). As the complement of a solution is also a solution, we can infer that there exists a clause such that it contains all but one variable from the solution.

For every clause in the given NAESPI we are going to try out every possible assignment of 1*0. There are at most $O(n^2)$ such possibilities. For a particular trial, let $P$ be the set of clauses which contain a variable assigned to 1 and $N$ the set of clauses which have a variable assigned to 0. All clauses which contain at least one variable set to 1 and at least one variable set to 0 are removed from further consideration because they are already satisfied.

It should be noted that each clause in $P \cup N$ contains exactly one variable assigned to 0 or 1, else there would be two clauses which intersect in more than one variable. Furthermore for any pair of clauses $p \in P$ and $q \in N$, $p$ and $q$ intersect only in the uninstantiated variables.
CHAPTER 3. THE BOUNDED CASE

As all the clauses in $N$ intersect in the variable which has been set to 0, no two uninstantiated variables in $N$ are the same, or we would have two clauses which intersect in more than one variable. Next we will try to put a lower bound on the number of uninstantiated variables in a clause in the set $P$.

**Lemma 3.3** For any clause $p \in P$, the number of uninstantiated variables is at least $|N|$.

**Proof:** $p$ has to intersect with every clause in $N$. The intersection cannot happen in the instantiated variables. Let $n_1, n_2 \in N$ be the two clauses which have a common intersection with $p$. Then $n_1$ and $n_2$ intersect in more than one variable, violating the fact that the given instance was 1-bounded.

Let $S$ be the solution which contains the variables set to 1 (denoted by $V$) in our trial. Let $U$ be the remaining set of variables in $S$. Let $U' \subseteq U$ be a minimal set such that $U'$ intersects with every clause in $N$. We claim that $V \cup U'$ is a candidate solution. As $U'$ is minimal, $U'$ intersects with every clause in $N$ in exactly one variable.

One naive algorithm now tries out every minimal set of variables in $N$ to determine if they define a solution. Let us assume that $|N| \geq 3$, else we can try out all the $O(n^2)$ minimal sets of variables which could be defining the solution. The next claim tells us how to compute a solution to 1-bounded NAESPI.

**Lemma 3.4** Let $|N| \geq 3$. Let $U_1$ be a minimal set of variables in $N$ which intersects with every clause in $N$. Let $U_2$ be another minimal set of variables such that $U_1$ and $U_2$ differ in exactly one variable. Then either $U_1 \cup V$ or $U_2 \cup V$ is a solution to the given NAESPI.

**Proof:** First, it should be noted that $|U_1| = |U_2| = |N|$. Also, any proper subset of $U_1$ or $U_2$ does not contain a clause in $P$, this follows from the fact that every clause in $P$ has at least $|N|$ uninstantiated variables (Lemma 3.3). Without loss of generality, assume that $U_1$ is not a solution. This implies that there exists a clause $p_1 \in P$ of type $\{1 \cup U_1\}$. If $U_2$ is also not a solution then there exists a clause $p_2 \in P$ of type $\{1 \cup U_2\}$. As $U_1$ and $U_2$ differ in only one variable, $|p_1 \cap p_2| \geq 2$, which violates the 1-bounded condition.

If there is only one minimal set $U_1$ which intersects every clause in $N$ then all the variables in $U_1$ are forced to have the same values as the variables in $V$ determining the solvability of the instance.
CHAPTER 3. THE BOUNDED CASE

If \(|N| \leq 2\) then we can exhaustively try out all the possible minimal sets which intersect clauses in \(N\).

**Lemma 3.5** 1-bounded \(NAESPI\) is solvable in \(O(n^4)\) time.

**Proof:** Correctness follows from Lemma 3.3 and Lemma 3.4. If \(|N| \leq 2\), we have to try out \(O(n^2)\) minimal sets as there are at most 2 clauses with at most \(n\) variables each. For each of the \(O(n^2)\) hitting sets it takes \(O(n^2)\) time to determine if the hitting set is a solution, hence the bound.

### 3.3.2 2-bounded \(NAESPI\)

Without loss of generality, assume that the minimal solution contains at least two variables, else the input instance is 1-solvable. Let \(a, b\) be the variables set to 1 in the minimal solution. This implies there are two clauses \(C_1 = (a \lor A), C_2 = (b \lor B)\), such that all the variables in the set \(A \cup B\) are set to 0 given the fact that \(a, b\) have been set to 1. Once again we can partition the set of clauses in the input into two sets: \(P\) denotes the set of clauses each of which has at least one variable set to 1 and \(N\) denotes the set of clauses each of which has at least one variable set to 0 in our trial. Clauses which contain variables set to both 1 and 0 will always be satisfied and are not considered further. All the clauses in \(P\) contain both \(a\) and \(b\) as they have to intersect with \(C_1\) and \(C_2\). Clauses in \(N\) contain no variable set to 1.

Assume \(|P| \geq 4\), else we can try out all the \(O(n^3)\) possibilities and determine the solvability of the instance in \(O(n^5)\) time, as there are \(O(n^3)\) possible solutions and determining if some subset of variables is a solution takes \(O(n^2)\) time.

**Lemma 3.6** Given \(N\) and \(P\) as defined above, the solvability of the input instance can be determined in polynomial time.

**Proof:** It should be noted that all the uninstantiated variables in the set of clauses \(P\) are distinct. This follows from the fact that all the clauses in the set \(P\) already intersect in the two variables \(a\) and \(b\). We are interested in finding a hitting set \(^1\) \(S\) of uninstantiated

---

\(^1\)The hitting set of a set of subsets is defined to be a set which contains at least one variable from every subset and does not contain any subset.
variables from $P$ such that $S$ does not contain any clause in $N$. If we have such a set $S$ then, setting $S$ to 0 and all the other variables to 1 leads to a solution.

Let $l$ be the minimum number of uninstantiated variables in a clause in $N$. This implies that $|P| \leq l$, else there are two clauses in $P$ which have an intersection in more than 2 variables. Furthermore every set of $(l-1)$ uninstantiated variables from the set of variables in $P$ does not contain any clause in $N$. This follows from the fact that $l$ is the cardinality of the minimum-sized clause.

Let $S_0$, $S_1$ be two hitting sets of clauses in $P$, such that $S_0$ and $S_1$ differ in exactly one variable. If two such hitting sets do not exist, then all the variables are forced to have an assignment different from the variables $a$ and $b$ and the solvability of the instance can be determined easily. As $S_0$ and $S_1$ differ in only one variable and $|P| \geq 4$, $|S_0 \cap S_1| \geq 3$.

This implies that either $S_0$ or $S_1$ does not contain a clause in $N$. If both $S_0$ and $S_1$ contained a clause in $N$ (note that each clause in $N$ has at least 3 variables), then there would be two clauses in $N$ which intersect in more than 2 variables. If $S_0$ is the hitting set which does not contain a clause in $N$, then setting all the variables in $S_0$ to 0 and the remaining variables to 1 leads to a solution to the input instance.

As there are $n$ clauses of size at most $n$, determining the right set of clauses $C_1$ and $C_2$ can take at most ($\binom{n}{2}$) and it takes $O(n^2)$ time to try out all the 10* possibilities. Hence the number of subproblems is $O(n^4)$. As it takes $O(n^2)$ time to verify whether some subset of variables is a solution, the total running time of the algorithm is $O(n^6)$. 

The case when $|P| \leq 3$ is treated in a similar way as the 1-bounded case. We try out all the $O(n^3)$ minimal sets of variables in the set $N$ which could be defining the solution. Since the verification takes $O(n^2)$ time, the total running time for the algorithm is $\max\{O(n^6),O(n^5)\}$, which is $O(n^6)$.

### 3.3.3 \(c\)-bounded NAESPI

In this section we will present a generalization of the results presented in the two previous sections. In particular we will show that \(c\)-bounded NAESPI can be solved in $O(n^{2c+2})$ time.

Given an instance of \(c\)-bounded NAESPI, without loss of generality, assume that the minimal solution contains $c+1$ variables at least, else the input instance is \(c\)-solvable and we
can determine the solution in $O(n^{c+2})$ time. This follows because there are at most $O(n^c)$ hitting sets which could be defining the solution and it takes $O(n^2)$ time to verify if some subset is extendible to a solution.

Let $\{a_1, \ldots, a_c\}$ be the variables in the minimal solution. This implies the existence of $c$ clauses $C_1 = (a_1 \lor A_1), C_2 = (a_2 \lor A_2), \ldots, C_c = (a_c \lor A_c)$, such that all the variables in the set $\cup_{i=1}^c A_i$ are set to 0 given the fact that the variables $a_1, a_2, \ldots, a_c$ have been set to 1. Once again we can partition the set of clauses in the input into two sets: $P$ denotes the set of clauses which have at least one variable set to 1 and $N$ denotes the set of clauses which have at least one variable set to 0 in our trial. Clauses which contain variables set to both 1 and 0 will be satisfied and are removed from further consideration. All the clauses in $P$ contain every $a_i$ as they have to intersect with every $C_i$. Clauses in $N$ contain no variable set to 1.

Assume, $|P| \geq c + 2$, else we can try out all the $O(n^{c+1})$ possibilities and determine the solvability of the instance in $O(n^{2c+2})$ time. Once again, there are at most $O(n^{c+1})$ hitting sets and for each hitting set we spend $O(n^2)$ time to verify if the hitting set is indeed a solution.

**Theorem 3.2** Given $N$ and $P$ as defined above, the solvability of the input instance can be determined in polynomial time.

**Proof:** It should be noted that all the uninstantiated variables in the set of clauses $P$ are distinct. We are interested in finding a hitting set $S$ of uninstantiated variables from $P$ such that $S$ does not contain any clause in $N$. If we have such a set $S$ then, setting $S$ to 0 and all the other variables to 1 leads to a solution.

Let $l$ be the minimum number of uninstantiated variables in a clause in $N$. This implies that $|P| \leq l$, else there are two clauses in $P$ which have an intersection in more than $c$ variables. Furthermore every set of $(l-1)$ uninstantiated variables from the set of variables in $P$ does not contain any clause in $N$. This follows from the fact that $l$ is the cardinality of the minimum-sized clause.

Let $S_0, S_1$ be two hitting sets of clauses in $P$, such that $S_0$ and $S_1$ differ in exactly one variable. If two such hitting sets do not exist, then all the variables are forced to have an assignment of values different from the variables $a$ and $b$ and the solvability of the
instance can be determined easily. As \( S_0 \) and \( S_1 \) differ in only one variable and \(|P| \geq c + 2\), \(|S_0 \cap S_1| \geq c + 1\).

This implies that either \( S_0 \) or \( S_1 \) does not contain a clause in \( N \). If both \( S_0 \) and \( S_1 \) contained a clause in \( N \) then there would be two clauses in \( N \) which intersect in more than \( c \) variables (note that each clause in \( N \) has at least \( c + 1 \) variables). If \( S_0 \) is the hitting set which does not contain a clause in \( N \), then setting all the variables in \( S_0 \) to 0 and the remaining variables to 1 leads to a solution to the input instance.

As there are \( n \) clauses of size at most \( n \), determining the right set of clauses \( C_1, \ldots, C_c \) and the \( 10^c \) assignments can take at most \( \binom{n^2}{c} = O(n^{2c}) \) time. As, it takes \( O(n^2) \) time to verify a solution, the total running time for this case is \( O(n^{2c+2}) \).

The case where \( |P| \leq c + 1 \) is treated in the same way as for the 2-bounded case. We try out all the \( O(n^{c+1}) \) minimal sets of variables in the set \( N \) which could be defining the solution. As it takes \( O(n^2) \) time to verify if some subset of variables is a solution and given the fact that there are at most \( O(n^{c+1}) \) hitting sets, the total running time of the algorithm is \( O(n^{2c+2}) \). Hence, the running time of the algorithm is dominated by \( O(n^{2c+2}) \).

### 3.4 Bounded number of pairs of clauses

In this section we parameterize the problem once again based on the number of clauses which have intersection of size exactly one. An instance of NAESPI is \((p-1)\)-bounded if the number of pairs of clauses which intersect in exactly one variable as at most \( p \). We show that \((p-1)\)-bounded NAESPI can be solved in \( n^{2p+2} \) time.

The following lemma follows for a given instance of NAESPI which is solvable.

**Lemma 3.7** Let \( c_1 \) and \( c_2 \) be two clauses which intersect in only one variable \( a \). Then there exists a pair of variables \( v_1 \in c_1 \) and \( v_2 \in c_2 \) such that \( v_1 \neq v_2 \neq a \) and \( v_1, v_2 \) get the same value in the solution.

**Proof:** To derive a contradiction, suppose that all the variables in \( c_1 \) except \( a \) are assigned different values from all the variables in \( c_2 \) except from \( a \) in any solution to the instance. Then, \( a \) is forced to be a 0 as well as a 1, leading to the desired contradiction.

In light of the above lemma for an instance which is \( p-1 \)-bounded, we can try out all the possible contractions for all the pairs of clauses. As there are at most \( n^2 \) contractions to be
performed for any given pair of clauses which intersect in one variable, the total number of tries is \((n^2)^p = n^{2p}\). After each such contraction all the pairs of clauses in the input instance intersect in at least two variables and the satisfiability can be determined in polynomial time by Lemma 2.4. As Lemma 2.4 takes \(O(n^2)\) time the running time of the algorithm is \(O(n^{2p+2})\).

3.5 Conclusion

In this chapter we studied c-bounded \(k\)-NAESPI and c-bounded NAESPI. For c-bounded \(k\)-NAESPI we showed that the number of clauses \(n \leq k^{c+1}\). We also showed that c-bounded \(k\)-NAESPI can be solved in \(O(n^{c+1}k)\) time. For c-bounded NAESPI we gave an \(O(n^{2c+2})\) algorithm for determining the satisfiability of the problem. We also showed that if there are at most \(p\) pairs of clauses which intersect in at most one variable then the NAESPI can be solved in \(O(n^{2p+2})\) time.
Chapter 4

A Quasi-polynomial time algorithm

4.1 Introduction

In this chapter we describe a quasi-polynomial time algorithm for solving the \textit{NAESPI} problem. Determining self-duality of monotone boolean functions can be used to solve the \textit{NAE-SPI} problem. Fredman and Khachiyan [19] describe a quasi-polynomial time algorithm for determining self-duality of monotone boolean functions. However, the algorithm we propose here is different from the algorithm of Fredman and Khachiyan and the analysis is also different.

4.2 Fredman and Khachiyan’s approach

First we briefly describe the approach of Fredman and Khachiyan [19] which tests if a pair of monotone boolean functions (in DNF) \( f = f(x_1, \ldots, x_m) \) and \( g = g(x_1, \ldots, x_m) \) are mutually dual. \( f \) and \( g \) are defined to be mutually dual if:\(^1\)

\[
f(x_1, \ldots, x_m) = \bar{g}(x_1, \ldots, x_m) \quad \text{for all } x \in \{0, 1\}^m.
\]

Duality of \( f(x) \) and \( g(x) \) is equivalent to the self-duality of the function \( h = yf(x) \lor
\]

\(^1\)Defering to the custom of denoting the input size by \( n \). We use \( n \) to denote the number of clauses, hence for the lack of a better symbol, the number of variables are denoted by \( m \).
$zg(x) \lor yz$ for two new variables $y$ and $z$. Hence their algorithm can also be used to test for self-duality of a monotone boolean function. Let $F$ and $G$ denote the set of prime implicants of $f$ and $g$ respectively. If $f$ and $g$ are mutually dual then for all $I \in F$ and $J \in G$ and with a slight abuse of notation (as the meaning is clear from the context)

$$I \cap J \neq \emptyset$$  \hspace{1cm} (4.1)

else, we can find a vector $z$ such that $f(x) = g(\bar{x}) = 0$. This is equivalent to saying that function $h$ is not dual-minor.

First they establish that if all the implicants of $f$ and $g$ are of size greater than $\log n$ (where $n = |F| + |G|$) then $f$ is not mutually-dual to $g$ and the vector satisfying $f(x) = g(\bar{x}) = 0$ can be obtained in polynomial time, where $n$ is the number of clauses in $f$ and $g$. This implies that there exists a logarithmically short implicant in $F$ (without loss of generality). Equation 4.1 now implies that there exists a variable which belongs to at least $\frac{n}{\log n}$ clauses. Next they look at Shannon's decomposition of $f$ and $g$, given by:

$$f = x_i f_0(y) \lor f_1(y), \quad g = x_i g_0(y) \lor g_1(y)$$  \hspace{1cm} (4.2)

where, $y = (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m)$, $f_0$ is the monotone irredundant DNF with implicant sets $F_0 = \{ I \mid i \in I, I \in F \}$, $f_1$ is the monotone irredundant DNF with implicant sets $F_1 = \{ I \mid i \notin I, I \in F \}$ and $g_0$ and $g_1$ defined similarly. It can be shown that $f$ and $g$ are mutually dual if and only if

$$f_1 \text{ is mutually dual to } g_0 \lor g_1$$  \hspace{1cm} (4.3)

and

$$g_1 \text{ is mutually dual to } f_0 \lor f_1$$  \hspace{1cm} (4.4)

The idea now is to recurse on the variable which belongs to $\frac{n}{\log n}$ clauses to obtain the decomposition described in Equation 4.2 and solve the subproblems in Equation 4.3 and Equation 4.4 recursively. They show that this simple scheme terminates in $O(n^{\log^3 n})$ time. Next they improve on the complexity of this algorithm using the following observation. Suppose that the subproblem described by Equation 4.3 has already been solved. Without
loss of generality we can assume that \( f_1 \) is dual to \( g_0 \lor g_1 \) else \( f \) is not dual to \( g \). Stated otherwise,

\[
f_1(\bar{y}) = \bar{g}_0(y) \bar{g}_1(y)
\]  
(4.5)

Equation 4.5 can now be used to simplify the subproblem associated with Equation 4.4. By substituting Equation 4.5 in Equation 4.4, we get

\[
g_1(y) = f_0(\bar{y}) \lor \bar{g}_0(y) \bar{g}_1(y).
\]  
(4.6)

It can be argued that Equation 4.6 has no solution when \( g_0(y) = 0 \). Hence, Equation 4.6 is equivalent to

\[
g_1(y) = f_0(\bar{y}) \land g_0(y) = 1.
\]  
(4.7)

Let \( G_0 \) be the set of prime implicants of \( g_0 \) and for \( J \in G_0 \), let \( y[J] \) be the vector obtained by substituting 1 for all the variables in \( J \). It can be argued that Equation 4.7 is equivalent to solvability of \( |G_0| \) equations

\[
g_1(y[J]) = f_0(\bar{y}[J]), J \in G_0
\]  
(4.8)

The algorithm now uses Equation 4.8 to solve the subproblem defined by Equation 4.4. Fredman and Khachiyan show that this modified algorithm terminates in \( n^{4\omega(\log n) + O(1)} \) where \( o(\log n) = \frac{\log n}{\log \log n} \).

In this thesis we reformulate the self-duality of monotone boolean functions as a special type of satisfiability problem. It was hoped that the dichotomy theorem of Schaefer [44] for the satisfiability problem would be applicable. In the next section we argue why Schaefer's result is not applicable to our problem.

### 4.3 Dichotomy theorem for satisfiability

As seen in Chapter 2, NAESPI can be modeled as a satisfiability problem in the following way: for every clause \( (x_1 \lor \ldots \lor x_k) \) in NAESPI we have two clauses in the satisfiability
problem, \((x_1 \lor \ldots \lor x_k)\) and \((\bar{x}_1 \lor \ldots \lor \bar{x}_k)\). It is easy to see that \(\text{NAESPI}\) is solvable if and only if the satisfiability problem has a satisfying assignment.

For an \(\text{NP-Complete}\) problem it is of interest to determine all the special cases that are polynomially solvable and prove that everything else is \(\text{NP-Complete}\). Schaefer [44] had the first such delineation result for the satisfiability problem. It identifies all the polynomially solvable subclasses of satisfiability (under some assumptions) and shows that the problem is \(\text{NP-Complete}\) otherwise. Similar results for the graph homomorphism problem have been obtained by Hell and Nešetřil [27].

We are given a finite set of logical relations \(S = (R_0, R_1, \ldots, R_t)\). For example \(R_0 = (x \lor \neg y \lor z)\). Relation \(R_0\) defines all the clauses which have exactly one negated literal in a clause containing three variables. An \(S\)-formula is any conjunction of the clauses each of type \(R_i \in S\). \(\text{SAT}(S)\) is the set of all satisfiable \(S\)-formulas.

**Theorem 4.1 (Schaefer 78):** \(\text{SAT}(S)\) is polynomially solvable (assuming \(P \neq NP\)) if and only if at least one of the following condition holds:

1. Every relation is \(S\) is satisfied when all the variables in it are set to 1.
2. Every relation is \(S\) is satisfied when all the variables in it are set to 0.
3. Every relation in \(S\) can be represented as a \(\text{CNF}\) formula which contains at most one negative literal.
4. Every relation in \(S\) can be represented as a \(\text{CNF}\) formula which contains at most one positive literal.
5. Every relation in \(S\) can be represented as a \(2\)-\(\text{CNF}\) formula.
6. Every relation in \(S\) is the set of solutions to a system of linear equations over field \(GF^2\).

If the dichotomy theorem for satisfiability were applicable to the \(\text{NAESPI}\) then it would be a strong indication that the \(\text{NAESPI}\) is polynomially solvable, as it is widely believed that \(\text{NP-Complete}\) and \(\text{Co-NP-Complete}\) problems cannot be solved in quasi-polynomial time. The reason why the dichotomy theorem is not applicable to \(\text{NAESPI}\) is because it assumes that the class of input instances \(I\) of satisfiability is closed under taking unions.
i.e., if \( I_1 \) and \( I_2 \) are two input instances from \( \mathcal{I} \) then \( I_1 \cup I_2 \) is also in \( \mathcal{I} \). But, NAESPI is not closed under the union operation.

### 4.4 Algorithm

Let \( C \) be an NAESPI problem. We denote the set of variables in \( C \) by \( V \) and the \( i^{th} \) clause by \( C_i \).

**Definition 4.1 (Partial assignment)** A partial assignment is a subset \( V' \subseteq V \) such that all the elements in \( V' \) are assigned a boolean value.

Given a partial assignment \( V' \), we divide the set of clauses \( C \) into three sets.

1. \( P \) denotes the set of clauses which have at least one variable set to 1 and no variable set to 0. \( P \) stands for the clauses with a positive prefix.
2. \( N \) denotes the set of clauses which have at least one variable set to 0 and no variable set to 1. \( N \) stands for the clauses with a negative prefix.
3. \( PN \) is the set of clauses which have at least one variable set to 1 and at least one variable set to 0.

It should be noted that \( P, N, PN \) are disjoint sets and the clauses in \( PN \) are satisfied as they contain at least one variable set to 1 and at least one variable set to 0.

**Definition 4.2 (Consistent partial assignment)** A partial assignment is said to be consistent if it is contained inside some solution to the NAESPI.

The next lemma shows that for a consistent partial assignment there is a clause either in \( P \) or in \( N \) which has a 'particular' type of assignment.

Next, we define the 'particular' type of assignment.

**Definition 4.3 (10*(01*) assignment)** A partial assignment is said to be of type 10*(01*) if exactly one variable is set to 1 and all the other variables are set to 0 (if exactly one variable is set to 0 and all the other variables are set to 1).
CHAPTER 4. A QUASI-POLYNOMIAL TIME ALGORITHM

Lemma 4.1 Given sets of clauses $P$ with respect to a consistent partial assignment $V'$, there exists a clause in $P$ which is satisfied as $01^*$ in the solution containing $V'$.

Proof: Let $S$ be the solution and all the clauses in $P$ have at least two variables set to 0. By changing one of the variables to 1 we decrease the number of 0's in this clause and no clause is unsatisfied, as clauses in $N$ already have a 0 assigned to them. We iterate until there is one clause with exactly one variable set to 0. □

Similarly,

Lemma 4.2 Given sets of clauses $N$ with respect to a consistent partial assignment $V'$, there exists a clause in $N$ which is satisfied as $10^*$ in the solution containing $V'$.

Proof: Similar to Lemma 4.1. □

Definition 4.4 (Unsafe variable in $N$) A variable $v \in N$ is said to be unsafe in $N$ if there exists a clause $C_v \in N$ containing $v$ such that setting $v$ to 1 and all the other variables to 0 satisfies at most half the clauses in $P$.

Symmetrically we can define unsafe variables in $P$.

Definition 4.5 (Unsafe variable in $P$) A variable $v \in P$ is said to be unsafe in $P$ if there exists a clause $C_v \in P$ containing $v$ such that setting $v$ to 0 and all the other variables to 1 satisfies at most half the clauses in $N$.

Next we describe an $O(n^{2\log n+2})$ algorithm for NAESPI. Our algorithm is a minor modification to the one presented in Chapter 2. We begin by trying out all the $10^*$ possibilities at the top level. For every choice made we compute the sets $P, N, PN$ and denote the resulting subproblem by $C_1$. Next we compute all the unsafe variables in $N$ (denoted $U$). Now we generate another subproblem $C_2$ obtained by collapsing all the variables in $U$. We solve $C_2$ recursively by trying out all the $10^*$ possibilities for clauses in $N$. We solve $C_1$ recursively but we only try those $10^*$ possibilities for which some variable in $U$ is set to 0. Formally,

Step 1 Let $P = N$ be the set of all the clauses and $NP = \emptyset$. 
CHAPTER 4. A QUASI-POLYNOMIAL TIME ALGORITHM

Step 2 If $P = \emptyset$ return satisfiable.

Step 3 Compute $U$, the set of unsafe variables in $N$.

Step 4 For every $u \in U$, solve the subproblem (obtained by setting $u = 0$) recursively.

Step 5 If all the subproblems in Step 4 fail then set all the variables in $U$ to 1 and solve all the subproblems (obtained by 10 assignments) recursively after removing all the satisfied clauses from $P$ and $N$.

Step 6 If all the subproblems fail at the top most level then return unsatisfiable.

Correctness of the algorithm follows from Lemma 4.1. To prove a bound on the running time we need the following lemmas. The next lemma states the obvious.

Lemma 4.3 If there are no unsafe variables in $N(P)$ then for every $10^p(01^p)$ trial at least half the clause in $P(N)$ are satisfied.

Proof: Follows from the definition of unsafe variables.

If $N(P)$ contains unsafe variables then we cannot guarantee that every trial of $10^p(01^p)$ in $N(P)$ reduces the number of unsatisfied clauses in $P(N)$ by half. Let $U$ denote the set of all the unsafe variables in $N(P)$. If all the variables in $U$ get assigned a value 1(0) for $U \in N(P)$ then we have satisfied at least one clause in $N(P)$. If one of the variables in $U$ gets assigned 0(1) then at least half the clauses in $P(N)$ are satisfied due to the fact that the variable is an unsafe variable. Let $C'$ be the problem obtained by contracting all the variables in $U \rightarrow u$. It should be noted that $C'$ has a solution if and only if all the variables in $U$ get assigned the same value in a solution to $C$. The next lemma puts a bound on the number of problems and the size of each subproblem generated for $C'$. Assume that $C'$ was obtained by contracting the variables in $N$. Let $S(n)$ denote the number of subproblems generated to solve a subproblem comprising $n$ clauses obtained after contracting all the unsafe variables $U$.

Lemma 4.4

$$S(n) \leq n^2S(\frac{|P|}{2})$$
CHAPTER 4. A QUASI-POLYNOMIAL TIME ALGORITHM

Proof: The total number of clauses is $n$ and each clause is of size at most $n$. Hence the total number of $10^*$ choices after collapsing all the variables in $U$ is at most $n^2$.

Note that the subproblems obtained do not have any unsafe variables in $N$. This is true because $U$ is the maximum set of unsafe variables in $N$. By Lemma 4.3, the size of $P$ is reduced by half for every subproblem. Hence,

$$S(n) \leq n^2 S\left(\frac{|P|}{2}\right)$$  \hfill (4.9)

Symmetrically, if $U$ was the set of unsafe variables in $P$ and the subproblems were obtained by contracting all the variables in $U$ then the next lemma holds.

Lemma 4.5

$$S(n) \leq n^2 S\left(\frac{|N|}{2}\right)$$

Proof: Similar to the proof for Lemma 4.4.

Theorem 4.2 The algorithm terminates in $O(n)^{2\log(|P|+2)}$ time.

Proof: Let $N$ and $P$ be the negative and positive sets of clauses in some stage $i$. Assume without loss of generality that $|P| \leq |N|$. Let $U$ be the set of unsafe variables in $N$. Assuming the instance is satisfiable, we have the following two cases:

- All the variables in $U$ are assigned a value of 1 in the solution. The collapsed problem $U \rightarrow u$ also has a solution. By Lemma 4.4 this requires no more than

$$S(n) \leq n^2 S\left(\frac{|P|}{2}\right)$$  \hfill (4.10)

subproblems to be solved.

- Some variable in $U$ is set to value 0 in the solution. In this case we try out at most $n^2$ possible $10^*$ assignments and for each such trial at least half the clauses in $P$ are satisfied. This is due to the fact the variable we set to 0 belongs to at least $|P|/2$ clauses (this is implied by the definition of unsafe variables). This requires no more than

$$S(n) \leq n^2 S\left(\frac{|P|}{2}\right)$$  \hfill (4.11)

subproblems to be solved.
Hence the total number of subproblems generated is at most \( O(n^{2\log|P|}) \). As it takes \( O(n^2) \) time to verify the solution, the total running time is \( O(n^{2\log|P|+2}) \).

\[ \square \]

### 4.4.1 Read-\( k \) boolean formulas

Read-\( k \) boolean functions are boolean formulas (in CNF) in which every variable belongs to at most \( k \) clauses. Read-\( k \) boolean formulas arise in the context of data mining and a host of other applications (for details see [9]). Read-\( k \) boolean formulas correspond to NAESPI instances in which every variable belongs to at most \( k \) clauses. Domingo, Mishra, and Pitt [9] show that given \( C \) a read-\( k \) formula in CNF and \( D \), a formula in DNF, equivalence of \( C \) and \( D \) can be determined in \( O(n^k) \) time. A corollary to Theorem 4.2 states that self-duality of read-\( k \) formulas in DNF can be determined in \( O(k^{O(\log k)}) \) time.

**Corollary 4.1** 
Self-duality of Read-\( k \) boolean formulas can be determined in \( O(k^{O(\log k)}) \) time.

**Proof:** Without loss of generality, we assume that there is a clause of logarithmic length, else the input is not self-dual. As each variable belongs to at most \( k \) clauses, the total number of clauses \( n \leq k \cdot \log n \). Therefore the time taken by the algorithm stated in the previous section is \( O(k \log n)^{\log n} \), which equals \( O(k^{\log n \log n^{\log n}}) = O((k^{\log n n^{\log n^{\log n}}}) \). By repeated substitution for \( n \) in the mantissa, we get \( O(k^{O(\log n)}) \). Using the same trick and substituting for \( n \) in the exponent, we get \( O(k^{O(\log k)}) \).

The previous corollary also implies that the equivalence of \( C \) and \( D \) (which are both read-\( k \) formulas in DNF) can be determined in \( O(k^{O(\log k)}) \) time, thereby solving a special case of [9] efficiently.

### 4.5 Variations of the algorithm

One obvious way to improve the complexity of the algorithm stated above is by showing that in any stage at most some constant number \( c \) of different 10\(^*\) trials are sufficient. Such an assertion, if true will yield a complexity of \( O(c^{\log n n^2}) \) for solving the NAESPI. Hence, the following question needs to be addressed.

\[ Q: \text{What is the maximum number of clauses which can be satisfied as 10\(^*\) for NAESPI?} \]
We give an infinite family of instances of NAESPI for which the number of clauses satisfied as $10^\omega$ at the top level is $o(\log n)$, where $n$ is the number of clauses in the input instance.

The example comprises three different sets of clauses $A, B, AB$ over the set of variables $\{a_1 \ldots a_k\} \cup \{b_1 \ldots b_k\}$. Clauses in set $A$ contain exactly one variable $a_i$, for some $i$ and all variables $b_1, \ldots, b_k$. Clauses in set $B$ contain exactly one variable $b_i$ and all the variables $a_1, \ldots, a_k$. Let $A'(B')$ be the set of all the subsets of size $\frac{k}{2} + 1$ of the set $A(B)$. Clauses in the set $AB$ are then the elements of the cross products of the sets $A'$ and $B'$. A concrete example for the case when the number of variables is eight is shown below.

The following lemma states that any pair of clauses in the example has a non-empty intersection.

**Lemma 4.6** Any pair of clauses has a non-empty intersection.

**Proof:** Let $C_1$ and $C_2$ be two clauses in the input. Consider the following cases.

1. $C_1, C_2 \in A$. The intersection happens in the variables $b_1, b_2, \ldots, b_k$.
2. $C_1, C_2 \in B$. The intersection happens in the variables $a_1, a_2, \ldots a_k$.
3. $C_1, C_2 \in AB$. The intersection happens in the variables $a_1, a_2, \ldots, a_k$ as every clause in $AB$ contains a $(\frac{k}{2} + 1)$-sized subset of the variables $a_1, a_2, \ldots, a_k$.
4. $C_1 \in A$ and $C_2 \in B$. The intersection happens in the variables $b_1, b_2, \ldots, b_k$ and the variable $a_1, a_2, \ldots, a_k$.
5. $C_2 \in A$ and $C_1 \in B$. This case is symmetric to the previous case.
6. $C_1 \in A$ and $C_2 \in AB$. The intersection happens in the variables $b_1, b_2, \ldots, b_k$.
7. $C_1 \in B$ and $C_2 \in AB$. The intersection happens in the variables $a_1, a_2, \ldots, a_k$.

An example for eight variables generated using the scheme outlined above follows:

**Example 4.5**

$$(a_1 \lor b_1 \lor b_2 \lor b_3 \lor b_4) \land$$
(a_2 \lor b_1 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_3 \lor b_1 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_4 \lor b_1 \lor b_2 \lor b_3 \lor b_4) \land \\
(b_1 \lor a_1 \lor a_2 \lor a_3 \lor a_4) \land \\
(b_2 \lor a_1 \lor a_2 \lor a_3 \lor a_4) \land \\
(b_3 \lor a_1 \lor a_2 \lor a_3 \lor a_4) \land \\
(b_4 \lor a_1 \lor a_2 \lor a_3 \lor a_4) \land \\
(a_1 \lor a_2 \lor a_3 \lor b_1 \lor b_2 \lor b_3) \land \\
(a_1 \lor a_2 \lor a_3 \lor b_1 \lor b_3 \lor b_4) \land \\
(a_1 \lor a_2 \lor a_3 \lor b_1 \lor b_2 \lor b_4) \land \\
(a_1 \lor a_2 \lor a_3 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_1 \lor a_2 \lor a_4 \lor b_1 \lor b_2 \lor b_3) \land \\
(a_1 \lor a_2 \lor a_4 \lor b_1 \lor b_3 \lor b_4) \land \\
(a_1 \lor a_2 \lor a_4 \lor b_1 \lor b_2 \lor b_4) \land \\
(a_1 \lor a_2 \lor a_4 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_1 \lor a_3 \lor a_4 \lor b_1 \lor b_2 \lor b_3) \land \\
(a_1 \lor a_3 \lor a_4 \lor b_1 \lor b_3 \lor b_4) \land \\
(a_1 \lor a_3 \lor a_4 \lor b_1 \lor b_2 \lor b_4) \land \\
(a_1 \lor a_3 \lor a_4 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_2 \lor a_3 \lor a_4 \lor b_1 \lor b_2 \lor b_3) \land \\
(a_2 \lor a_3 \lor a_4 \lor b_1 \lor b_3 \lor b_4) \land \\
(a_2 \lor a_3 \lor a_4 \lor b_1 \lor b_2 \lor b_4) \land \\
(a_2 \lor a_3 \lor a_4 \lor b_2 \lor b_3 \lor b_4) \land \\
(a_2 \lor a_3 \lor a_4 \lor b_2 \lor b_4) \lor \\
(a_2 \lor a_3 \lor a_4 \lor b_4)
Lemma 4.7 Given a solution the number of clauses satisfied as $10^*$ is $o(\log n)$.

Proof: Setting the variables $a_1, \ldots, a_k$ to 1 and the variables $b_1, \ldots, b_k$ to 0 is a solution to the input instance. Such a solution satisfies only the clauses in sets $A$ and $B$ as $10^*$. All the clauses in the set $AB$ have at least two variables in the solution. The number of clauses in the set $AB$ is

$$\left(\frac{k}{2} + 1\right) \times \left(\frac{k}{2} + 1\right)$$

which is $O(k^k)$. Choosing $k \sim \log n$, we get that the number of clauses satisfied as $10^*$ is $o(\log n)$.

Consider the following variation of the algorithm proposed above. In each stage the clause which has to be tried is picked with probability $\frac{1}{2}$. The previous example also settles the non-existence of such a randomized probabilistic algorithm with a better running time.

It might still be possible to select the right clauses which have to be tried as $10^*$ deterministically. One approach would be to try clauses which intersect in at most 1 variable. This sounds like a reasonable strategy because if all the clauses intersect in at least 2 variables then the input is trivially satisfiable due to Lemma 2.4. The next lemma states that even this modified strategy for selecting the clauses can be fooled by a minor modification to the example presented above. We define set $AB$ to the set of clauses which contains the elements of cross product of $A'$ and $\{b_1, b_2\}$.

Lemma 4.8 The number of disjoint pairs of clauses which intersect in at most 1 variable is $O(k^k)$.

Proof: For every $a_1, a_2 \in A'$ there are two clauses in $AB$ such that the two clauses intersect in exactly one variable. The number of such pairs is $\binom{|A'|}{2}$ which is $O(k^k)$. ■

4.6 Conclusion

In this chapter we presented a simple algorithm for determining the satisfiability of the NAESPI problem. We showed that the algorithm has a running time of $O(n^{2\log n+2})$. We gave an infinite class of instances for which the number of clauses which are satisfied as $10^*$ is at most $o(\log n)$, thereby ruling out an obvious modification to the algorithm in hope of decreasing the running time.
4.7 Note

As pointed out by Kaz Makino the algorithm is extensible to the non-monotonic case, where the literals are not constrained to be positive.
Chapter 5

Average Case Analysis

5.1 Introduction

In this chapter we examine the average case behaviour of a variant of the algorithm presented in Chapter 4 on a class of randomly generated NAESPI instances. Considerable research has been conducted in analyzing the average case behaviour of various algorithms for the satisfiability problem. Two models which have been used extensively for generating instances for the satisfiability problem are the constant density model and the constant component model.

According to the constant density model \( M_1(n, r, p) \), each of the \( n \) clauses is generated independently from the \( r \) boolean variables, where each literal belongs to a clause with probability \( p \). In the constant component model \( M_2(n, r, k) \), each of the \( n \) clauses is selected independently and uniformly from the set \( L \) of all \( k \)-literal clauses over \( r \) variables.

Goldberg, Purdom and Brown [25, 42] show that for instances generated using the constant density model, the Davis-Putnam procedure runs in polynomial time. But the constant density model has been criticized [18] because it allows 0-literal clauses with almost the same probability that it allows \( k \) literal clauses. Hence, the random instances generated according to \( M_1(n, r, p) \) are either unsatisfiable with a high probability or satisfiable by a random assignment with a high probability. Franco [18] argues that the constant density

\[1\] 0-literal clauses are the empty clauses.
model is therefore inferior to the constant component model.

For the purpose of analyzing the average case behaviour we can use the constant density model for generating the problem. However, for our purposes this generation mechanism does not suffice, as the intersection amongst the clauses is not guaranteed.

Another way of generating instances of NAESPI would be to use Bioch and Ibaraki’s scheme [4] for generating all the self-dual functions, which in CNF representation would correspond to the NAESPI instances which are unsatisfiable. Satisfiable instances of NAESPI could then be obtained by removing a clause from the unsatisfiable instances.

An NAESPI instance is symmetric if all the variables belong to the same fraction of the total number of clauses and this property holds for every subset of clauses in the given instance. Symmetric instances are satisfiable in polynomial time by Khachiyan’s algorithm and a variant of our basic algorithm (to be described in Section 5.3). A cursory look reveals that all the self-dual functions of seven variables are symmetric. Due to the lack of a rigorous study, at this juncture we cannot rule out the possibility that this is a not a good method of generating a test-bed of instances.

In the next section we propose a method of randomly generating NAESPI instances.

### 5.2 Random NAESPI

The *Auxiliary graph* defined in Chapter 3 can also be used to generate random instances of the NAESPI problem. To generate an instance of NAESPI with $n$ clauses and $m$ variables, we consider a clique of size $n$ (with $\binom{n}{2}$ edges). Every variable belongs to an edge with probability $p$ and does not belong to an edge with probability $1 - p$. If $L_v$ is the set of variables on the edges incident on node $v$, then clause $v$ is defined to be $L_v$.

This generation scheme guarantees that the intersection property is obeyed, but it can generate instances in which there are clauses containing some other clause. As our algorithm is insensitive to the containment of clauses this seems like a viable method for generation. For any two clauses, the corresponding vertices in the graph share the variables that belong to the edge between the two vertices, thus guaranteeing intersection.

Our generation scheme can be visualized as a union of $m$ edge labelled random graphs over the same set of vertices, such that each random graph has a label associated with all its
edges (a label corresponds to a variable). The two most studied models of random graphs are $G(n, p)$ and $G(n, e)$, defined below.

**Definition 5.1 (G(n,p))** Given a positive integer $n$, $G(n, p)$ is the probability space on the graphs over $n$ vertices with edge probability $p$ where the events are mutually independent.

In other words, if $G_e$ is a graph over $n$ vertices with exactly $e$ edges then

$$P[G = G_e] = p^e(1 - p)^{n-e}$$

Another widely studied model of random graphs is the class $G(n, e)$.

**Definition 5.2 (G(n,e))** $G(n, e)$ comprises all the graphs over $n$ vertices containing exactly $e$ edges where each graph is equally likely.

From a theorem of Angluin and Valiant [6] it follows that the two models are practically interchangable when $e$ is close to $pn$. Hence, we will consider only the $G(n, p)$ model.

Let $V_i$ be the indicator variable whose value determines if the $i^{th}$ vertex has been covered by some edge in a randomly generated graph $G$. $E_j$ is the indicator variable which determines if the $j^{th}$ edge has been chosen.

Let $G = (n, e)$ be a random graph over $n$ vertices with exactly $e$ edges where $e \leq n$. If all the edges in $G$ are equally likely then the average number of vertices covered in $G$ is $O(e)$. 

**Lemma 5.1** $Pr[|V| \leq k] \geq Pr[|E| \leq \frac{k}{2}]$.

**Proof:** The claim follows from the fact that if $\frac{k}{2}$ edges are present then at most $k$ vertices are covered. 

Let $G = (n, e)$ be a random graph over $n$ vertices with exactly $e$ edges where $e \leq n$. If all the edges in $G$ are equally likely then the average number of vertices covered in $G$ is $O(e)$. 


Lemma 5.2 If the number of vertices covered in $G$ is at most $n\frac{\sqrt{5} - 1}{2}$ then the expected number of vertices covered satisfies $\mathbb{E}[V] \geq \frac{\sqrt{5} - 1}{2}e$.

Proof: If $x$ is the fraction of vertices covered by some $e' < e$ edges then the probability that we increase the number of vertices covered for each edge chosen is at least $1 - x^2$. $|V| \leq n\frac{\sqrt{5} - 1}{2}$ implies that $x \leq \frac{\sqrt{5} - 1}{2}$, and therefore the probability of success is at least $1 - (\frac{\sqrt{5} - 1}{2})^2$. This implies that the expected number of vertices covered is at least $\frac{\sqrt{5} - 1}{2}e$. ■

Next we use the Chebyshev inequality to bound the tail probability of Lemma 5.2. Without loss of generality we assume that the probability with which we increase the number of vertices covered is $\frac{\sqrt{5} - 1}{2}$. Our experiment can also be visualized as flipping a coin $e$ times where the probability of head occurring (denoted $p$) in each individual trial is $\frac{\sqrt{5} - 1}{2}$. This is the binomial distribution with mean $\mu_x = ep$ and standard deviation $\sigma_x = \sqrt{e\bar{p}q}$. For the next lemma we assume that $p = \frac{\sqrt{5} - 1}{2}$ and $q = 1 - p$.

Lemma 5.3 If $e$ edges are covered then with probability $1 - \frac{4e}{ep}$ at least $\frac{ep}{2}$ vertices are covered.

Proof: The Chebyshev inequality asserts that,$$Pr[|X - \mu_x| > t\sigma_x] \leq \frac{1}{t^2}$$choosing $t\sigma_x = \frac{ep}{2}$, we get $\frac{1}{t^2} = \frac{4e}{ep}$. Hence, with probability $1 - \frac{4e}{ep}$ we cover at least $ep - \frac{ep}{2} = \frac{ep}{2}$ vertices. Observe that this probability goes to 1 as $e \to \infty$. ■

The probability that $k$ edges are chosen (for a given label) is given by the binomial distribution.$$Pr[E = k] = \binom{m}{k}p^k(1 - p)^{m-k}$$where $m = \binom{n}{2}$. Next, we use the Demoivre-Laplace theorem [16] to compute the probabilities we are interested in.

Theorem 5.1 (DeMoivre-Laplace) Suppose $0 < p < 1$ depends on $n$ in such a way that $npq \to \infty$ as $n \to \infty$. If $0 < h = x\sqrt{pqn} = o(pqn)^{\frac{3}{2}}$ and if $x \to \infty$, then $Pr[|X - np| > h] \sim \frac{e^{-\frac{h^2}{2}}}{x\sqrt{2\pi}}$. 

The DeMoivre-Laplace theorem is obtained by approximating the binomial distribution with the normal distribution. For a random variable $X$ it gives the probability that $X - np \geq x$, where $x$ is measured in steps of $\sqrt{npq}$, where $np$ is mean and $\sqrt{npq}$ is the standard deviation of the binomial distribution.

**Lemma 5.4** Let $m = \binom{n}{2}$. For $p \leq \frac{\log n}{m}$, $Pr[E \leq 2 \log n] \geq 1 - \frac{1}{\sqrt{2\pi}\log ne^{(\log n/2)}}$.

*Proof:* Let $p = \frac{\log n}{m}$ then $\mu_x = mp = \log n$ and $\sigma_x^2 = mpq = \log n(1 - \frac{\log n}{m}) \sim \log n$ for large $m$. Choosing $h = \log n$, we get $x = \frac{\log n}{\log n} = \sqrt{\log n}$.

Then by the DeMoivre-Laplace theorem,

$$Pr[E \geq 2 \log n]_p \sim \frac{1}{\sqrt{2\pi}\log ne^{(\log n/2)}}$$

and for $p' < p$

$$Pr[E \leq \log n]_{p'} \geq Pr[E \leq \log n]_p$$

Hence, the result.

Similarly,

**Lemma 5.5** Let $m = \binom{n}{2}$. For $p \geq \frac{1}{n}$, $Pr[E \geq \frac{n}{4}] \geq 1 - \frac{4}{\sqrt{2\pi}\log ne^{(\log n/2)}}$.

*Proof:* Let $p = \frac{1}{n}$ then $\mu_x = mp = \frac{n-1}{2}$ and $\sigma_x^2 = mpq = n(1 - \frac{1}{n}) \sim n$ for large $n$. Choosing $h = \frac{n}{4}$, we get $x = \frac{n}{4\sqrt{n}} = \frac{\sqrt{n}}{4}$.

Then by the DeMoivre-Laplace theorem,

$$Pr[E \leq \frac{n}{4}]_p \leq \frac{4}{\sqrt{2\pi}\log ne^{(\log n/2)}}$$

and for $p' > p$

$$Pr[E \geq \frac{n}{4}]_{p'} \geq Pr[E \geq \frac{n}{4}]_p$$

Hence, the result.

In the next section we describe a simple modification to the algorithm presented in Chapter 4. The modification is based on the observations of Fredman and Khachiyan [19].
5.3 A Variant

We describe a variant of the algorithm we described in Chapter 4. To recap, we begin by trying out all the $10^n$ possibilities at the top level. For every choice made we compute the sets $P, N, PN$. We denote the resulting subproblem by $C_1$. Next we compute all the unsafe variables in $N$ (denoted $U$). Now we generate another subproblem $C_2$ obtained by collapsing all the variables in $U$. We solve $C_2$ recursively by trying out all the $10^n$ possibilities for clauses in $N$. We solve $C_1$ recursively but we only try those $10^n$ possibilities for which some variable in $U$ is set to 0.

The first key observation is that we do not need to try $10^n$ possibilities for all the variables in a given stage. Let $v$ be a variable which is unassigned in Stage $i$. In what follows we will change the definition of an unsafe variable slightly.

**Definition 5.3 (Unsafe in $P(N)$)** A variable is called unsafe in $P(N)$ if it belongs to more than $\frac{n}{\epsilon}$ clauses in $P(N)$ for some $\epsilon$ respectively, where $n$ is the total number of clauses in $P$ and $N$.

The following three cases are of interest to us and form the basis for the variant which we develop in this section.

i) If $v$ is unsafe in both $P$ and $N$ then we just try setting $v$ to 0 and 1. For each trial we get a subproblem of size $n(1 - \frac{1}{\epsilon})$. Previously we were generating $O(n)$ subproblems when trying a value of 1 for the variable $v$. As the variable is either a 1 and 0 in the solution we move to Stage $i+1$ after solving these two subproblems.

ii) If $v$ is unsafe in $P$ and safe in $N$ then we first try setting $v$ to 1, which results in a subproblem of size at most $n - 1$. If we fail then $v$ occurs as a $01^*$ in the solution. Here, we generate at most $n$ subproblems each of size at most $\frac{n}{\epsilon}$. This follows from the fact that for each $01^*$ trial we have to consider only those clauses in $N$ which contain the variable $v$. All the other clauses in $N$ will already have a 1 assigned, because of the intersection property.

The case when $v$ is unsafe in $N$ and safe in $P$ is symmetric.

iii) When $v$ is safe in both $P$ and $N$ is treated in the same way as the previous case.
The modification to our algorithm comprises of tackling these three cases separately as advocated by Fredman and Khachiyan. The correctness of the modified algorithm still follows from Theorem 4.2 and the bound on the running time follows from the next theorem.

**Theorem 5.2 (Fredman & Khachiyan:96)** The algorithm terminates in $n^{4o(\log n) + O(1)}$ time.

**Proof:** Here we will give only an outline of their proof. Fredman and Khachiyan choose $\epsilon = \log n$ thereby obtaining a bound of $n^{4o(\log n) + O(1)}$. They also argue that the third case never happens as there is always a variable which is unsafe either in $P$ or in $N$. Otherwise all the clauses in $P$ and $N$ are of size greater than $\log n$. Hence, the running time for the algorithm is given by the following two recurrences:

- The first case is described by the following recurrence,

  $$ f(n) = 2f(n(1 - \frac{1}{\epsilon})) + 1 $$

- This corresponds to the second case shown above. Complexity of this recurrence is dominated by the second term.

  $$ f(n) = f(n - 1) + nf(\frac{n}{\epsilon}) $$

They show that both these recurrences are bounded by $n^{\frac{4\log n}{\log \log n} + O(1)}$.

We conclude that by using the insight of Fredman and Khachiyan it is possible to improve the performance of the algorithm presented in Chapter 4 to achieve a running time of $n^{4o(\log n) + O(1)}$ by making the modifications described at the start of this section.

In the next section we show that Fredman and Khachiyan's algorithm plus the variant of our algorithm based on their approach terminates in

$$ \max\{O(n^{3.87}), O(n^{\log^2 n}), O(\frac{1}{p} - \frac{1}{\sqrt{p}})^{o(\log \frac{p}{\sqrt{p}})}\} $$

time for the class of random NAESPI.
5.4 Analysis

In this section we analyze the average case behaviour of the algorithm presented in the previous section. The key idea is to use Lemmas 5.3, 5.4 and 5.5.

**Theorem 5.3** The algorithm presented in the previous section terminates in
\[ \max\{O(n^{3.87}, O(n^3 \log^2 n), O\left(\frac{1}{p} - \frac{1}{\sqrt{p}}\right)^{2+o(\log \frac{1}{p}\sqrt{p})}\} \]

time on average, where \( p \) is the probability with which the initial input instance is generated.

**Proof:** Let \( P_k(N_k) \) denote the number of clauses in sets \( P(N) \) at Stage \( k \). The following cases are of interest:

i) Before some stage \( i \), \( p \geq \max\{\frac{1}{P^k}, \frac{1}{N_k}\} \). By Lemma 5.5 and Lemma 5.3 there exists a variable which belongs to at least \( P_j \sqrt{\frac{N_k}{4}} \) and \( N_i \sqrt{\frac{N_k}{4}} \) clauses. Here, the first case of the algorithm is invoked, which generates two subproblems each of size at most \( O(P) \) or \( O(N) \). The number of subproblems is described by the following recurrence:
\[ f(n) = 2f(n \frac{5 - \sqrt{5}}{4}) \]
which is \( O(n^{1.87}) \). As it takes \( O(n^2) \) time to verify the solution for each subproblem, the total running time is \( O(n^{3.87}) \).

ii) At some stage \( j \geq i \) either \( p \leq \frac{\log P_j}{P_j^2} \) or \( p \leq \frac{\log N_j}{N_j^2} \). By Lemma 5.4 and Lemma 5.3 there exists a variable which belongs to at most \( \log n \) clauses in \( P_j \). Therefore the number of subproblems of size \( \frac{P_j}{\log P_j} \) generated in the second step of the algorithm is at most \( \log n \) and the number of subproblems is given by the following recurrence:
\[ f(n) = nf(\log n) \]
which is \( O(n \log^2 n) \). As it takes \( O(n^2) \) time to verify the solution for each subproblem, the total running time is \( O(n^3 \log^2 n) \).

iii) Without loss of generality assume that \( \frac{\log P_j}{P_j^2} \leq p \leq \frac{1}{P_j} \). The time taken in this case by Theorem 5.2, is at most \( O(P_i - P_j)^{o(\log P_i - P_j)} \). Given that \( \frac{1}{P_j^2} \leq \frac{1}{P_i} \), we obtain
\[ P_j \geq \sqrt{P_i} \]. Substituting the lower bound on \( P_j \) in the previous equation, the time
taken is at most $O(P_i - \sqrt{F_i})^o(\log P_i - \sqrt{F_i})$. Also, we have $p \leq \frac{1}{P_i}$, equivalently $P_i \leq \frac{1}{p}$. Hence, the number of subproblems in this case is at most $O(\frac{1}{p} - \frac{1}{\sqrt{p}})^o(\log \frac{1}{p} - \frac{1}{\sqrt{p}})$. As it takes $O(n^2)$ time to verify the solution for each subproblem, the total running time is $O(\frac{1}{p} - \frac{1}{\sqrt{p}})^2 + o(\log \frac{1}{p} - \frac{1}{\sqrt{p}})
$.

The following is a corollary to Theorem 5.3.

**Corollary 5.1** If $p \geq \frac{1}{(\log n)\log \log n}$ then the algorithm terminates in $O(n^{2\log \log n + 2})$ time.

**Proof:** Suppose that $\frac{1}{P_i} \leq p$ in some stage $i$ then there are at most $(\log n)^{\log \log n}$ clauses remaining. At this point the algorithm (the one presented in Chapter 4) takes at most $O(n^{2\log \log n + 2})$ time. Hence, the result. \(\blacksquare\)

### 5.5 Conclusion

In this chapter we proposed a method for randomly generating NAESPI instances. We presented a variation of our basic algorithm for solving NAESPI (the one presented in the previous chapter) based on the ideas of Fredman and Khachiyan [19]. We showed that the variant of our basic algorithm has the same complexity as the algorithm of Fredman and Khachiyan [19], i.e., $O(n^{4(\log n) + O(1)})$. Furthermore, we analyzed the performance of our algorithms on the randomly generated instances of NAESPI and showed that the algorithm terminates in max\{$O(n^{3.87}), O(n^3 \log^2 n), O(\frac{1}{p} - \frac{1}{\sqrt{p}})^2 + o(\log \frac{1}{p} - \frac{1}{\sqrt{p}})$\} time.
Chapter 6

Approximation Algorithm for NAESP

6.1 Introduction

In this chapter we study a general version of the NAESPI problem and describe an approximation algorithm for solving the same. We remove the restriction that each pair of clauses has a non-empty intersection. A polynomial time approximation algorithm is a good way of coping with an NP-Complete problem because it takes polynomial time and the solution is guaranteed to be 'not too far from the optimum'. The notion of 'not too far' is formalized next.

Let \( \Pi \) be an optimization problem. For each instance \( p \in \Pi \) we have a set of feasible solutions \( F(p) \). Let \( c(s) \) be the value associated with a solution \( s \in F(p) \). Then, the optimum value for instance \( p \) is defined as

\[
OPT(p) = \max_{s \in F(p)} c(s)
\]

for a maximization problem. For a minimization problem the optimum value is

\[
OPT(p) = \min_{s \in F(p)} c(s)
\]

Let \( A \) be an algorithm for \( \Pi \) such that for input \( p \) it returns a feasible solution \( A(p) \in F(p) \).
$F(p)$. $A$ is an $\epsilon$-approximation algorithm if for all $p$, we have

$$\frac{|c(A(p)) - OPT(p)|}{\max \{OPT(p), c(A(p))\}} \leq \epsilon$$

MAX-CUT is the problem of partitioning the nodes of an undirected graph $G = (V, E)$ into two sets $S$ and $V - S$ such that there are as many edges as possible between $S$ and $V - S$. MAX-CUT is NP-Complete [41]. However the problem of determining a cut of minimum size can be solved in polynomial time by using the algorithm of Edmonds and Karp [13], which builds on the max-flow min-cut theorem established in [17, 15].

NAESP is the not-all-equal satisfiability problem with only positive literals in the clauses where intersection is not required between pairs of clauses. MAXNAESP is defined to be the problem of determining the maximum number of clauses which can be satisfied. MAXNAESP is a generalization of the MAX-CUT problem which is evident from the following reduction. Given a graph $G = (V, E)$, for every edge $e = (u, v) \in E$ we have a clause $(u, \mathbb{I})$ in the corresponding MAXNAESP problem. It is easy to see that if there exists a cut of size $s$ then there exists a solution to the NAESP problem which satisfies at least $s$ clauses. Hence, we can regard MAXNAESP as a generalization of MAX-CUT.

The technique of local improvement has been widely used as a heuristic for solving combinatorial optimization problems. A novel approximation algorithm for MAX-CUT is based on the idea of local improvement [41]. The idea is to start with some partition $S$ and $V - S$ and as long as we can improve the quality of the solution (the number of cross edges) by adding a single node to $S$ or by deleting a single node from $S$, we do so. It has been shown that this approximation algorithm for MAX-CUT has a worst case performance ratio of $\frac{1}{2}$ [22].

In this chapter we describe another approximation algorithm based on local improvement for solving MAXNAESP which yields a performance ratio of $\frac{4}{s+1}$ where each clause contains at least $s$ variables. The algorithm described gives a bound of $\frac{2}{3}$ for MAX-CUT in contrast to the bound of $\frac{1}{2}$ achieved by the local search algorithm described in [22].

We give a polynomial reduction from 3NAES, in which each clause has exactly three literals, and the literals are not restricted to be positive [21], to NAESP, thereby establishing the NP-Completeness of NAESP. In Section 2 we look at MAXNAESP and provide two approximation algorithms for it.
We begin with some definitions:

**Definition 6.1 (NAES)** Given a set of clauses, find a satisfying truth assignment such that each clause contains at least one true literal and one false literal.

**Definition 6.2 (NAESP)** Given a set of clauses with only positive literals, find a satisfying truth assignment such that each clause contains at least one true literal and one false literal.

Given an instance of 3NAES (which is known to be NP-Complete [21]) we will generate an instance of 3NAESP. NP-Completeness of MAX-2-NAESP (where each clause contains exactly two variables) follows from the fact that it is equivalent to MAX-CUT. The next lemma is similar to Lemma 2.8.

**Lemma 6.1** 3NAES is polynomially equivalent to 3NAESP.

*Proof:* Let \( n \) be the number of clauses and \( m \) be the number of variables in 3NAES. Let us denote the literals in 3NAES by \( x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_m, \overline{x_m} \). We replace each \( \overline{x_i} \) with a new literal \( x_{m+i} \) in all the clauses. In addition, for each pair of literals \( x_i, \overline{x_i} \), we add the four clauses \((x_i, x_{m+i}, a), (x_i, x_{m+i}, b), (x_i, x_{m+i}, c), (a, b, c)\). We have a total of \((n + 4m)\) clauses in the instance of 3NAESP so generated.

\( \Rightarrow \) If 3NAES is satisfiable then the set of clauses generated in the instance of 3NAESP generated is also satisfiable. The solution is obtained by assigning \( x_{n+i} \) the same value as \( \overline{x_i} \) in 3NAES. In addition, the literal \( a \) is set to 0 and the literals \( b, c \) are set to 1.

\( \Leftarrow \) If the clauses in the instance of 3NAESP generated is satisfiable then the solution to 3NAES is obtained by setting \( \overline{x_i} \) to the same value that \( x_{n+i} \) is set to. This assignment clearly satisfies all the clauses. For literals \( x_i, \overline{x_i} \), the four clauses \((x_i, x_{m+i}, a), (x_i, x_{m+i}, b), (x_i, x_{m+i}, c), \text{ and } (a, b, c)\) guarantee that both \( x_i \) and \( \overline{x_i} \) are not set to 1 (or 0).
6.2 Approximation Algorithms

6.2.1 $\frac{1}{2}$ Approximation Algorithm

We first present a simple approximation algorithm for MAXNAESP which has a worst-case performance bound of $\frac{1}{2}$. We then examine a locally optimal approximation algorithm for the problem whose worst-case performance bound is $\frac{2}{3}$ for the problem, and in general has a bound of $\frac{s}{s+1}$, where $s$ is the minimum number of literals in a clause.

$\frac{1}{2}$ Approximation Algorithm: Let the problem comprise $m$ literals and $n$ clauses. A literal $u$ is arbitrarily picked and set to 1 if it occurs in at least $\frac{m}{2}$ clauses. The remaining literals are set to 0 and the algorithm terminates. If however the literal $u$ occurs in only $s < \frac{m}{2}$ clauses, then the literal $u$ is set to 0 and the subproblem comprising the $n - s$ clauses not containing the literal $u$ is recursively solved (with the literal $u$ removed from the subproblem). If the solution $S$ to the subproblem satisfies more than $\frac{s}{2}$ clauses containing $u$ then the algorithm returns the solution $S$ for the remaining literals. Else the algorithm returns $\overline{S}$, the complement of the solution $S$ for the remaining literals (in the complement solution $\overline{S}$, all the literal settings are complemented).

**Theorem 6.1** The above algorithm has a performance ratio of $\frac{1}{2}$.

**Proof:** We prove by induction on the number of literals $m$ that the algorithm satisfies at least $\frac{m}{2}$ clauses.

**Base Case:** For $m = 2$ (the minimum number of literals required for NAESP), each clause contains both literals (else the clauses cannot be satisfied and can therefore be removed) and so the result follows trivially.

**Induction Hypothesis:** Assume the algorithm satisfies at least $\frac{1}{2}$ the total number of clauses for $m \leq p$ literals.

**Induction Step:** Let the number of clauses be $n$. Let the set of clauses satisfied by literal $u$ be denoted $U$. The algorithm either sets $u$ to 1 (if $u$ satisfies $s \geq \frac{n}{2}$ clauses), or sets $u$ to 0 and solves the subproblem with $p - 1$ literals (and $n - s$ clauses) recursively. By the induction hypothesis, the algorithm derives a solution $S$ that satisfies at least $\left\lfloor \frac{n-s}{2} \right\rfloor$ clauses in the subproblem. If in addition $S$ also satisfies at least $\frac{s}{2}$ clauses among $U$, then the algorithm satisfies at least $\frac{n}{2}$ in total. If on the other hand $S$ satisfies less than $\frac{s}{2}$ clauses
among $U$, then $\overline{S}$ satisfies at least $\frac{s}{2}$ clauses among $U$. This follows from the fact that the clauses that are unsatisfied in $U$ due to assignment $S$ have all their variables set to 0, implying that the assignment $\overline{S}$ satisfies all these clauses (because the variable $u$ in all these clauses is set to 0). In addition, both the assignment $S$ and its complement $\overline{S}$ continue to satisfy the same number of clauses in the subproblem. Thus the algorithm satisfies at least $\frac{s}{2}$ clauses in total.

A trivial example where a literal satisfies exactly $\frac{1}{2}$ the total number of clauses suffices to show that the bound is tight.

In the next section we describe an approximation algorithm for MAXNAESP and show that the algorithm has a performance ratio of $\frac{s}{s+1}$, where $s$ is the minimum clause length.

6.2.2 $\frac{s}{s+1}$ Approximation Algorithm

This approximation algorithm relies on the notion of a local optimum.

**Definition 6.3 (Locally optimal solution)** A solution $S$ to MAXNAESP, is locally optimal if the number of clauses satisfied cannot be increased by complementing the value to which a literal is set in the solution $S$.

The algorithm starts with a random assignment of values to literals (say all literals are set to 0). It then complements the setting of a literal if this improves the solution. The algorithm continues in this manner until no further improvement results when a literal setting is complemented.

Let $S$ be the locally-optimal solution. With respect to this solution $S$, we divide the set of literals into two disjoint subsets, $X = \{x_1, x_2, \ldots, x_p\}$ comprising all literals which are set to 1, and $Y = \{y_1, y_2, \ldots, y_q\}$, comprising all literals which are set to 0. We divide the clauses of MAXNAESP into three sets, $A$, $B$, and $C$. $A$ is the set of all satisfied clauses, $B$ ($C$) is the set of all unsatisfied clauses for which each literal in a clause belongs to $X$ ($Y$). We note that each clause in $A$ contains at least one literal that belongs to $X$, and one literal that belongs to $Y$ (else the clause will not be satisfied). Let $B(x_i), 1 \leq i \leq p$ ($C(y_j), 1 \leq j \leq q$) denote the number of clauses in $B$ ($C$) which contain the literal $x_i$ ($y_j$). Let $A(x)$ ($A(y)$) denote the number of clauses in $A$ that contain only one literal from the set $X$ ($Y$).
Lemma 6.2 \( A(x) \geq \sum_{i=1}^{p} B(x_i) \).

**Proof:** For the \( B(x_i) \) clauses in \( B \) containing the literal \( x_i \), there should be \( A(x_i) \geq B(x_i) \) clauses in \( A \) which contain only literal \( x_i \) from the set \( X \) (and all other literals from the set \( Y \)). If this is not the case then by setting \( x_i \) to 0 we can increase the number of clauses satisfied, violating the fact that \( S \) is locally-optimal. Noting that a clause in \( A \) which contains only literal \( x_i \) is distinct from a clause in \( A \) that contains only literal \( x_j \), \( i \neq j \), \( 1 \leq i, j \leq p \), it follows that the \( A(x_i) \) clauses in \( A \) containing only literal \( x_i \) are distinct from the \( A(x_j) \) clauses in \( A \) containing only literal \( x_j \). Thus, \( A(x) = \sum_{i=1}^{p} A(x_i) \geq \sum_{i=1}^{p} B(x_i) \). \( \blacksquare \)

The corollary below may be shown using similar reasoning as above.

**Corollary 6.1** \( A(y) \geq \sum_{j=1}^{q} C(y_j) \).

Lemma 6.3 below derives an upper bound on \( |B| \).

**Lemma 6.3** \( |B| \leq \sum_{i=1}^{p} \frac{B(x_i)}{s} \).

**Proof:** Each clause \( b_i \in B \) has \( s_i \geq s \) literals in it. Counting the number of 1's occurring in the clauses in set \( B \), we can write \( \sum_{i=1}^{p} B(x_i) = \sum_{i=1}^{|B|} s_i \). The left hand side counts the occurrence of 1's in each literal, and the right hand side counts the occurrence of 1's in each clause. But \( \sum_{i=1}^{|B|} s_i \geq \sum_{i=1}^{|B|} s = |B| \times s \), from which the result follows. \( \blacksquare \)

By a similar reasoning, the corollary below follows.

**Corollary 6.2** \( |C| \leq \sum_{j=1}^{q} \frac{C(y_j)}{s} \).

**Theorem 6.2** Let \( A \) be an algorithm which returns a locally-optimal solution to the MAX-NAESP problem. The performance ratio of \( A \) is given by \( r_p \geq \frac{s}{s+1} \).

**Proof:** It follows from the definition that \( |A| \geq A(x) + A(y) \), since \( A \) is the set of all clauses that are satisfied, and \( A(x) \) (\( A(y) \)) is the set of clauses that are satisfied with exactly one literal from the set \( X \) (\( Y \)). From Lemmas 6.2 and 6.3, and Corollaries 6.1 and 6.2, it follows that \( |A| \geq A(x) + A(y) \geq \sum_{i=1}^{p} B(x_i) + \sum_{j=1}^{q} C(y_j) \geq s \times (|B| + |C|) \). \( |A| \geq A(x) + A(y) \)
because $A$ is the set of clauses which are satisfied and $A(x)$ is the set of clauses which are satisfied as $10^*$ and $A(y)$ the set of clauses which are satisfied as $01^*$ (refer to Chapter 2 for definition of clauses satisfied as $10^*$ and $01^*$). Hence the performance ratio $r_p$ is given by,

$$r_p = \frac{|A|}{(|A|+|B|+|C|)} \geq \frac{(|B|+|C|)^s}{(|B|+|C|)^s+|B|+|C|} \geq \frac{s}{s+1}.
$$

The following example illustrates that the above bound is tight. The instance contains three clauses: $u_1 \lor u_3$, $u_1 \lor u_2$, and $u_2 \lor u_4$. A locally optimal solution assigns $u_1 = u_2 = 1$, and $u_3 = u_4 = 0$, satisfying two clauses, when the optimal assigns $u_1 = u_4 = 1$, and $u_2 = u_3 = 0$, satisfying three clauses. This example can be extended for general $s$, where each clause has at least $s$ literals.

### 6.3 Conclusion

As the main result in the chapter, we propose a simple locally optimal approximation algorithm for the MAXNAESP problem whose performance ratio is $\frac{s}{s+1}$, where $s$ is the minimum number of literals in a clause. Such an algorithm appears to be useful for a large class of problems in the general domain of satisfiability. Specifically, we note that this algorithm has a bound of $\frac{s}{s+1}$ for MAXSAT, and follows directly from Lemmas 6.2 and 6.3. This bound is identical to the bound derived by Johnson for a simple greedy algorithm [41].

It should be noted that the NAESP problem where each clause has exactly 2 literals is equivalent to the MAX-CUT problem. However, a simple adaptation of the semi-definite approximation algorithm for solving the max-cut problem [24] for MAXNAESP (where each clause has at least two literals) does not appear to work. In addition, a straight-forward extension of the randomized rounding algorithm (using the linear-programming solution as the probabilities) [23] yields a bound of $\frac{1}{2}$. Also, the simple randomized algorithm, where each literal is selected with probability $\frac{1}{2}$ has an expected bound of $1 - \frac{1}{2^{s+1}}$, where $s$ is the minimum clause length. In contrast to this, the algorithm proposed in this chapter obtains a bound of $\frac{s}{s+1}$ for the MAXNAESP problem. Though the randomized algorithm has a better expected bound for $s \geq 4$, the algorithm proposed in this chapter obtains a bound of $\frac{2}{3}$ (as compared to $\frac{1}{2}$ for the randomized algorithm), for the most general version of the MAXNAESP problem, obtained when $s = 2$. 
Chapter 7

Conclusion and Open problems

In this thesis we established the equivalence between NAESPI and Self-Duality. We describe an $O(n^{2\log n+2})$ algorithm for solving the NAESPI problem and showed that the bound is tight for the given algorithm. We showed that an adaptation of our algorithm based on the observations of Fredman and Khachiyan [19] has the same complexity as their algorithm. We provided an average-case analysis of the adapted algorithm for randomly generated instances of NAESPI. The average-case running time was shown to be $\max\{O(n^{3.87}), O(n^3\log^2 n), O(\frac{1}{p} - \frac{1}{\sqrt{p}})^{2+o(\log \frac{1}{p} - \frac{1}{\sqrt{p}})}\}$, where $p$ is the probability with which the instance is generated. We showed that $k$-NAESPI and $c$-bounded NAESPI can be solved in polynomial time. We described a linear time algorithm for $k$-NAESPI. We also showed that finding strong solutions to the NAESPI problem is NP-Complete. We addressed the relationship between easily solvable instances of NAESPI and almost self-dual functions [5]. We also described approximation algorithms for the NAESP problem (where the intersection property is relaxed). NAESP is a generalization of the MAX-CUT problem and is equivalent to 2-coloring of hypergraphs.

The following problems are open to the best of our knowledge. The linear time algorithm (Chapter 2) for solving $k$-NAESPI can be used to solve $\sqrt{\log n}$-NAESPI in $n^2 \sqrt{\log n}$ time, which leads to the following:

**Open Problem 2** Is $(\log n)$-NAESPI solvable in polynomial time?

The example (presented in Chapter 4) shows an instance with at most $\log n$ clauses
satisfied as $10^n$ admits an easy solution. The following question begs some serious consideration.

**Open Problem 3** Does there exist a class of non-easily satisfiable NAESPI instances such that at most $\log n$ clauses are satisfied as $10^n$?

A related question is:

**Open Problem 4** Can the satisfiability of non-easily satisfiable NAESPI instances be determined in polynomial time?
Appendix A

List of Symbols

$C$ denotes a Coterie.

\textbf{$c$-bounded NAESPI} is NAESPI such that every pair of clauses intersect in at most $c$ variables.

\textbf{$c$-bounded $k$-NAESPI} is $k$-NAESPI which is $c$-bounded.

$\mathbb{E}[V]$ denotes the expectation of the random variable $V$.

$f$ denotes a Monotone Boolean Functions in DNF.

$k$-NAESPI is NAESPI with at most $k$ variables in every clause.

$K_i$ denotes a clique of size $i$.

$m$ is the number of variables in the function $f$.

$n$ usually refers to the size of the input, for example the number of clauses in NAESPI.

$N$ is the set of clauses such that all the instantiated variables in every clause are set to 0.

$P$ is the set of clauses such that all the instantiated variables in every clause are set to 1.

$P(f)$ is the Not All Equal Satisfiability problem with intersection (NAESPI) obtained from the monotone boolean function $f$. 
$PN$ is the set of clauses such that each clause contains at least one variable set to 1 and at least one variable set to 0.

$Q_i$ denotes the $i^{th}$ quorum in a Coterie.

$s$ is the cardinality of the minimum-sized clause for a given NAESP.

$S$ refers to a solution to the problem under consideration.

$\bar{S}$ is the complement of a solution.

$S(n)$ denotes the number of subproblems generated to solve an instance of NAESPI with $n$ clauses.

$T$ denotes a term in $f$.

$T(n)$ denotes the time taken by some algorithm on an input of size $n$.

*uniform* NAESPI is NAESPI such that every clause contains the same number of variables $k$, for some $k$.

$z_i$ usually refers to the $i^{th}$ variable (monotone) in the given instance of NAESPI.
Bibliography


BIBLIOGRAPHY


