2006

Horn formula minimization

Tom Chang

M.S. Project Report

Tom Chang
(tyc0429@cs.rit.edu)

May 7, 2004

Acknowledgement Throughout this project and my graduate studies here in RIT, I have received help from numerous sources. I would like to take this chance to express my gratitude. First I want to thank Professor Edith Hemaspaandra, without whose guidance, advice, patience and encouragement, this project would not have been possible. I also want to thank Professor Stanislaw P. Radziszowski and Professor Christopher Homan for taking the time to review and critique my work, and Professor Hans-Peter Bischof for guiding me and believing in me throughout the years. I want to thank the Laboratory for Applied Computing for financially supporting me through my project. Lastly, I want to thank my family and friends here in Rochester for supporting me and for always being there.

Contents

1 Abstract

Horn formulas make up an important subclass of Boolean formulas that exhibits interesting and useful computational properties. They have been widely studied due to the fact that the satisfiability problem for Horn formulas is solvable in linear time. Also resulting from this, Horn formulas play an important role in the field of artificial intelligence.

The minimization problem of Horn formulas is to reduce the size of a given Horn formula to find a shortest equivalent representation. Many knowledge bases in propositional expert systems are represented as Horn formulas. Therefore the minimization of Horn formulas can be used to reduce the size of these knowledge bases, thereby increasing the efficiency of queries.

The goal of this project is to study the properties of Horn formulas and the minimization of Horn formulas. Topics discussed include

- The satisfiability problem for Horn formulas.
- NP-completeness of Horn formula minimization.
- Subclasses of Horn formulas for which the minimization problem is solvable in polynomial time.
- Approximation algorithms for Horn formula minimization.

2 Introduction

Boolean functions and Boolean formulas representing them are basic building blocks of logic, and Horn functions and Horn formulas are essential tools in knowledge base representation and query handling. It has long been established that the satisfiability problem for Horn formulas is solvable in polynomial-time [9], but the more complex problem of minimization remains NP-complete for Horn formulas.

The Horn minimization problem, simply put, is to find a minimal representation that is equivalent to a given Horn formula. The size of a Horn formula is often measured in the number of clauses or the number of literal occurrences. Since propositional expert systems are often implemented using Horn formulas, minimizing a Horn formula effectively reduces the size of a knowledge base, which results in improved performance of the expert system.

In this paper, in addition to studying the problem of Horn minimization, we also study the relevant problem of satisfiability, and we classify subclasses of Horn formulas that can be minimized in polynomial-time. We present some results from prior papers, including two proofs of the NP-completeness of Horn minimization, one reducing from the problem of finding a minimal Set Covering, and one reducing from the well known problem of Hamiltonian Path. Two different approximation algorithms from prior papers are also presented, where one reduces a Horn formula into a prime and redundantly Horn formula, and the other iteratively applies some switching of variables, and separates a Horn formula into a minimized part and one that can be further minimized. Our own results include classifying the subclasses of Horn formulas for which minimization is in polynomial time, and providing a polynomial time algorithm for each case. The algorithm for 2-Horn formula minimization (Horn formulas where each clause has at most 2 literals) is shown to also minimizes general 2-CNF formulas.

The rest of this paper is organized as follows: Section 3 introduces the necessary terminologies and tools for the discussion. Section 4 discusses the problem of Boolean formula satisfiability. Section 5 discusses the minimization problem for general Boolean formulas, and Section 6 discusses the minimization problem for Horn formulas. Sections 7, 8, 9, 10, and 11 discuss subclasses of Horn formulas for which minimization is in polynomial-time. Finally Section 12 presents two approximation algorithms in prior papers for the problem of Horn formula minimization.

3 Basic terminology

3.1 Boolean functions

A Boolean function f of n propositional variables $p_1, \ldots, p_n$ is a mapping of assignments of the variables $\{0,1\}^n$ to truth values $\{0,1\}$. Every truth assignment sets each of the Boolean variables $p_1, \ldots, p_n$ to either *true* or *false*, and the function f evaluates to either *true* or *false* with each truth assignment of the variables. Boolean functions can be represented as truth tables, where each row denotes one possible truth assignment. A function with n Boolean variables has 2^n possible truth assignments, therefore the size of truth tables grows exponentially as the number of variables increases. Boolean functions with more than a handful variables are more commonly expressed as Boolean formulas.

3.2 Boolean formulas

It is common knowledge that every Boolean function can be represented by a Boolean formula that is built from propositional variables, the operators AND, OR, NOT, and the constants *true* and *false*. In a Boolean formula, variables $p_1, \ldots, p_n$ and their negations $\overline{p_1}, \ldots, \overline{p_n}$ are called positive and negative literals respectively. Conjunctions of literals are formed using the AND ($\wedge$) operator. And disjunctions of literals are formed using the OR ($\vee$) operator. A clause or term is a conjunction or a disjunction of literals. Boolean formulas in conjunctive normal form (CNF) are conjunctions of disjunctive clauses, such as $(p_1 \vee \overline{p_2} \vee p_3) \wedge (\overline{p_4} \vee p_5)$. Likewise, Boolean formulas in disjunctive normal form (DNF) are disjunctions of conjunctive clauses, such as $(p_1 \wedge p_2 \wedge \overline{p_3}) \vee (p_4 \wedge p_5)$. A Boolean function can be represented by different Boolean formulas. For instance, $(a \wedge b) \vee (b \wedge c)$ and $(\overline{a} \wedge b \wedge c) \vee (a \wedge b \wedge \overline{c}) \vee (a \wedge b \wedge c)$ both represent the same function. Such different formula representations of the same Boolean function are called equivalent to each other.

It is also well known that every Boolean function can be expressed as a formula in CNF, and as a formula in DNF. The most trivial way to turn a Boolean function into a DNF formula is to create a conjunctive clause out of each unique truth assignment that evaluates to *true*. Each variable in the particular truth assignment with the value *true* will be in the corresponding conjunctive clause as positive literals, and the rest negative literals. A DNF of

the function is then created by forming a disjunction with all such conjunctive clauses. A CNF formula for a Boolean function can be derived by first negating the outcome of each truth assignment, then creating a DNF out of the negated function using the above technique. Finally, the DNF can be transformed into a CNF of the original function by applying DeMorgan's laws.

The minimization problem, simply put, is to find the shortest formula that represents the same function as a given formula. The minimization problem of CNF formulas is computationally equivalent to the minimization problem of DNF formulas. This is because the negation of a Boolean formula does not alter the length of the formula in the number of literal occurrences or the number of clauses. Given a Boolean formula in DNF to be minimized, we can first negate the formula to get an CNF of the negated function by DeMorgan's law, minimize the CNF, then negate the minimized version to get a minimized DNF back. Therefore an algorithm that minimizes Boolean formulas in CNF can be used to minimize Boolean formulas in DNF and vice versa. Because it is cumbersome and inconvenient to continue the discussion covering both CNF and DNF formulas, for the rest of this paper, we will restrict ourselves to CNF formulas only. Therefore, from this point on, when we speak of a clause, we mean by it a disjunctive clause like ones that make up a CNF formula.

3.3 Horn formulas

A clause is called Horn if it contains at most one positive literal, and it is called pure Horn or definite Horn if it contains exactly one positive literal. A CNF formula is called Horn if every one of its clauses is Horn, and is called pure or definite Horn if every one of its clauses is pure Horn. Below are some examples of Horn CNF formulas:

- Horn CNF: $(p_1 \vee \overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_4} \vee \overline{p_5}) \wedge (p_6 \vee \overline{p_7})$
- Definite Horn CNF: $(\overline{p_1} \vee p_2 \vee \overline{p_3}) \wedge (p_4 \vee \overline{p_5} \vee \overline{p_6})$

A Boolean function is called Horn if it has at least one Horn formula representation. Furthermore, a Boolean function is called definite Horn if it has at least one definite Horn formula representation.

Horn formulas are used extensively in logic programming because the satisfiability problem for Horn formulas can be solved in polynomial-time.

In fact, as it will be shown in Section 4.3, the satisfiability problem for Horn formulas is solvable in linear time. This special characteristic of Horn formulas makes it possible to efficiently compute logic inferences in large logic databases. In logic programming, only three forms of relations are allowed:

1. Goals, such as $\leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$
2. Facts, such as $a \leftarrow$
3. Rules, such as $a \leftarrow b_1 \wedge \ldots \wedge b_n$

These translate directly into disjunctive Horn clauses $(\overline{b_1} \vee \overline{b_2} \vee \ldots \vee \overline{b_n})$, (a_1) and $(a \vee \overline{b_1} \vee \ldots \vee \overline{b_n})$. The logic database would consist of numerous clauses of these forms. When an query is made, the logic system constructs a Horn CNF by taking the conjunction of relevant clauses and the conditions of the query, and proceeds to solve the satisfiability problem for the resulting formula.

3.4 Implicates of a Boolean function

Given two Boolean functions f and g, if for every possible truth assignment, $f(p_1, \ldots, p_n) = true$ implies $g(p_1, \ldots, p_n) = true$, then we denote this relationship by $f \rightarrow g$, meaning f implies g. Likewise, for two clauses C and C', we say that $C \rightarrow C'$ if C implies C'. For example, given clauses $C = (p_1 \vee p_2)$ and $C' = (p_1 \vee p_2 \vee p_3)$, $C \rightarrow C'$ because every truth assignment that makes C *true* makes C' *true*. Now if $f \rightarrow C$ where f is a function and C a clause, we say that C is an *implicate* of f. For example, the clause $C = (p_1 \vee p_2)$ is an implicate of $f = (p_1 \vee p_2) \wedge (p_2 \vee p_3)$. An implicate C of a function f is called *prime* if there is no other implicate C' of f such that $C' \rightarrow C$. Therefore even though $C = (p_1 \vee p_2 \vee p_3)$ is an implicate of $f = (p_1 \vee p_2) \wedge (p_2 \vee p_3)$, it is not a prime implicate because $C' = (p_1 \vee p_2)$ is an implicate of f such that $C' \rightarrow C$. On the other hand, the clause $(p_1 \vee p_2)$ is a prime implicate because no other implicates of f implies $(p_1 \vee p_2)$. Since dropping any literal from an implicate C always produces a clause that implies C, an implicate C of f is prime if and only if dropping any literal from C results in a clause that is not an implicate of f.

It is easy to see that every clause in a CNF is a implicate of the function represented by the CNF, but not every clause in a CNF is necessarily prime. We call a CNF *prime* if every clause in the CNF is prime. A prime implicate

of a Boolean function is called *essential* if it is contained in every prime CNF of the function. Finally, a CNF is called *irredundant* if every clause in the CNF is required to represent the function. A redundant clause C in a CNF F is one where the resulting CNF after C is removed, $F \backslash C$, is equivalent to F. An irredundant CNF is a CNF that does not contain a redundant clause.

3.5 Resolution

Resolution is a technique that is often used in proofs in propositional logic. It was introduced by Robinson in 1965 [19], where he showed it to be sound, and as an inference rule for refutation, complete. We will explain the soundness and completeness of resolution as it applies to Boolean formulas as we illustrate resolution with an example.

Resolution is based on the basic inference rule:

$$((p \vee F_1) \wedge (\overline{p} \vee F_2)) \rightarrow (F_1 \vee F_2)$$

Essentially, resolution produces a new clause from two existing clauses by joining a clause with a positive literal of some variable with another clause with the negative literal of the same variable to form a new clause without the presence of literals of this variable.

The soundness of resolution means that the implication $((p \vee F_1) \wedge (\overline{p} \vee F_2)) \rightarrow (F_1 \vee F_2)$ always holds, in other words, it is a tautology. This is easy to see. It follows then that the new clause is an implicate of the two original clauses. This means that every new clause obtained by performing resolution on a CNF formula is an implicate of the formula.

An inference rule is complete if every statement implied by a given formula can be generated by repeatedly applying the inference rule. Resolution by itself is not complete, because not all implicates of a given Boolean formula can be derived using resolution. However, it is refutation complete, meaning that when resolution is used as an inference rule to derive refutation in propositional calculus, it is complete. This is often stated like the following lemma:

Lemma 3.1 *If F is an unsatisfiable CNF, then the empty clause can be derived from F using resolution [19].*

The completeness of resolution refutation means that even though we are not able to generate every implicate of a CNF formula, we are able to

generate every prime implicate of a CNF formula by performing resolution repetitively. This property can be stated as the following

Corollary 3.2 *If F is a satisfiable CNF and C is a prime implicate of F, then C can be derived by resolution from F.*

Proof Suppose $C = (p_1 \vee p_2 \vee \ldots \vee p_n)$, then $\overline{C} = \overline{(p_1 \vee p_2 \vee \ldots \vee p_n)} = (\overline{p_1} \wedge \overline{p_2} \wedge \ldots \wedge \overline{p_n})$, and $F \wedge \overline{C} = F \wedge \overline{p_1} \wedge \overline{p_2} \wedge \ldots \wedge \overline{p_n}$.

Since C is an implicate of F, $F \wedge \overline{C}$ is an unsatisfiable formula. Then according to Lemma 3.1, the empty clause can be derived by resolution from $F \wedge \overline{C}$. We know that since F is satisfiable, the empty clause cannot be derived by resolution from the clauses in F alone. Moreover, since C is a prime implicate, we know that dropping any literal from C results in a clause that is not an implicate of F. Therefore, the resolution steps that produce the empty clause from $F \wedge \overline{C}$ must resolve with every clause $\overline{p_1}, \overline{p_2}, \ldots, \overline{p_n}$ from $\overline{C}$. Otherwise if the empty clause can be derived from resolving with only a proper subset of these clauses, then the disjunction of the complements of said clauses would be an implicate of F that can be formed by dropping some literals from C, thus contradicting the primality of C.

To derive the empty clause from the clauses $\overline{p_1}, \overline{p_2}, \ldots, \overline{p_n}$, we must be able to derive $(p_1 \vee p_2 \vee \ldots \vee p_n)$ from F. Therefore, C can be derived by resolution from F. ∎

4 SAT

4.1 General SAT

The satisfiability problem for Boolean formulas (SAT) is a famous problem that has been well studied. The decision problem of SAT is stated as the following:

> Given a Boolean formula F, does it have a truth assignment such that F evaluates to *true*?

It is obvious that SAT is in NP. If someone claimed that a formula is satisfiable and gave us a truth assignment of the variables as the certificate, we can easily evaluate the value of the formula in polynomial-time. Solving

SAT exhaustively is in EXP though, as an algorithm with exponential running time ($O(2^n)$, where n is the number of variables) can simply try every single truth assignment and report if a satisfying assignment is found.

In 1972, Cook established SAT for CNF formulas (CNF-SAT) as the first NP-complete problem by proving that any problem that can be solved by an NP Turing machine can be many-one reduced in polynomial-time to a Satisfiability problem of a CNF formula [6]. Here we define some crucial terms:

Definition 4.1 Polynomial-time reduction*: Language A is polynomial-time reducible to language B, if there exists a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that for every string w,*

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the polynomial-time reduction *of A to B.*

Definition 4.2 NP-hardness*: A language B is NP-hard if every langauge A in NP is polynomial-time reducible to B.*

Notice that an np-hard problem may be beyond the class of NP. If a problem in NP is np-hard, then we have the following definition:

Definition 4.3 NP-completeness*: A language B is NP-complete if B is in NP, and every language A in NP is polynomial-time reducible to B.*

The Cook-Levin Theorem proved that CNF-SAT every problem in NP can be polynomial-time reduced to CNF-SAT. This means that the complexity of solving CNF-SAT is as hard as solving any problem in NP. By the same token, if CNF-SAT could be reduced to another problem in NP, then that problem is as hard as CNF-SAT, therefore as hard as any problem in NP. Therefore, CNF-SAT was the cornerstone on which the class of NP-complete problems was founded.

Since Cook's Theorem, SAT for subclasses of Boolean formulas have been studied. The most complete result was that of Schaefer's dichotomy in 1978 [20], which stated the following:

Theorem 4.4 (Schaefer's Dichotomy Theorem) *For every finite set S of logical relations, the satisfiability problem of SAT(S) is either polynomial-time decidable or NP-complete, and it is polynomial-time decidable if and only if it meets one of the following conditions:*

- *Every relation in S is satisfied when all variables are 0. (0-valid)*
- *Every relation in S is satisfied when all variables are 1. (1-valid)*
- *Every relation in S is definable by a CNF formula in which each conjunct has at most one negated variable. (Horn)*
- *Every relation in S is definable by a CNF formula in which each conjunct has at most one unnegated variable. (anti-Horn)*
- *Every relation in S is definable by a CNF formula having at most 2 literals in each conjunct. (affine)*
- *Every relation in S is the set of solutions of a system of linear equation over the two element field 0,1. (bijunctive)*

Here the definition of a logical relation is any subset of $\{0,1\}^k$, and SAT(S) is the satisfiability problem for a CNF formula that is the conjunction of a number of clauses. Each of these CNF formulas represents a logical relation in the set S.

Obviously, the class of Boolean functions most relevant to this project is that of Horn functions. Even before Schaefer published his results, SAT for Horn formulas was shown to be decidable in polynomial-time [16]. In 1984, Dowling and Gallier presented a linear time algorithm for determining the satisfiability of Horn formulas [9]. These algorithms are discussed in greater detail in the Section 4.3.

4.2 2-SAT

Before looking at SAT for Horn formulas, let us first look at another subclass of CNF formulas for which SAT is in P. 2-SAT is the satisfiability problem for CNF formulas with at most 2 literals per clause. This is the bijunctive case from Schaefer's Dichotomy Theorem. The fact that 2-SAT can be solved in polynomial-time will be needed in Section 9. Formally stated, the problem of 2-SAT is as follows:

2-SAT: Given a CNF formula F, where each clause in F has at most 2 literals, does there exist a truth assignment to the variables in F such that F evaluates to *true*?

2-SAT has been shown to be solvable in polynomial-time in [6, 2, 18], and shown to be solvable in linear time in [10, 3, 7]. Here we present two different algorithms, the first from [3], because we use the construction of this algorithm later on in Section 9. The second algorithm is from [7], which is a standard 2-SAT solver.

In [3], an implication graph is constructed from a given 2CNF formula. The implication graph construction and properties of the graph is discussed in further detail in Section 9. For now, it suffices to note that an arc in the implication graph $u \rightarrow v$ means that the implication $u \rightarrow v$ exists in the function represented by the 2CNF formula, meaning that $(\overline{u} \vee v)$ is an implicate of the given 2CNF formula. Below we present the graph building algorithm and the satisfiability checking algorithm.

Given a 2-CNF F:

Construct a directed graph $G(F)$ where
For every variable $x \in F$, add 2 nodes, x and $\overline{x}$ $(\overline{\overline{x}} \equiv x)$
For every clause $(a \vee b) \in F$, add arcs $(\overline{a}, b)$ and $(\overline{b}, a)$

In $G(F)$, each vertex v and its complement vertex $\overline{v}$ correspond to the variable v and its complement $\overline{v}$ in F. Therefore, the vertices in $G(F)$ may be assigned truth values just like variables in a Boolean formula. And just like a variable and its complement in a Boolean formula, a vertex v and its complement vertex $\overline{v}$ must receive complementary truth values. Moreover, the arc $a \rightarrow b$ in $G(F)$ denotes the logical implication $a \rightarrow b$ in F. Therefore, just as the logical implication $a \rightarrow b$ asserts that if a is *true*, then b must be *true*, in the implication graph $G(F)$, the truth assignment of the vertices must not lead from a true vertex to a false vertex. Then the theorem below follows immediately.

Theorem 4.5 *(Theorem 1 of [3]) A 2CNF formula F is satisfiable if and only if no vertex v in the implication graph $G(F)$ is in the same strongly connected component as its complement vertex $\overline{v}$.*

Given a 2CNF implication graph, the satisfiability testing can be implemented using a graph traversing algorithm such as depth first search, where if a vertex and its complement are found in the same strongly connected component, then the formula is unsatisfiable. For a formula with n variables

and m clauses, the graph construction runs in $O(n+m)$, and the satisfiability testing runs also in $O(n+m)$.

The second satisfiability testing algorithm, from [7], is a restricted case of the Davis Putnam procedure [8]. The general Davis Putnam procedure is an exhaustive search method based on resolution that determines whether a given CNF is satisfiable. This 2-SAT algorithm and uses the same principle as the Davis Putnam procedure, and uses unit resolution to propagate partial truth assignments. Unit resolution is a restricted form of resolution where one of the resolving clauses is always a unit clause. The pseudo code for the algorithm is presented below:

```
Given a 2CNF formula F:

Procedure BTOSat(F)
F := PropUnit(F);
while □ ∉ F and F is not empty
   choose an unassigned variable x
   P := PropUnit(F ∧ (x));
   if □ ∈ P
      F := PropUnit(F ∧ (x̄));
   else
      F := P;
if □ ∈ F
   return "unsatisfiable";
else
   return current assignment;

Procedure PropUnit(F)
while F contains an unit clause U
   if F contains Ū
      F := (F\(U)\(Ū)) ∧ □;
   else
      for each clause C in F
         if C contains U
            F := F\{C};
         else if C contains Ū
            F := F\{C} ∧ {C\Ū};
      F := F\(U);
```

```
return F;
```

This algorithm is called "backtrack once" (BTOSat), and uses a "guess and deduce" logic to find whether a 2CNF formula is satisfiable. Since a unit clause in a CNF must evaluate to *true* for the CNF to be satisfied, PropUnit simply propagates truth assignments to variables resulting from making a unit clause *true*. As can be seen from the pseudo code, BTOSat chooses a variable arbitrarily to assign a *true*, then propagates this assignment using unit propagation. If assigning this variable *true* results in a contradiction, then this variable is assigned *false* instead. If assigning a variable *true* and assigning it *false* both yield a contradiction, then the formula is unsatisfiable. BTOSat backtracks at most once, and runs in $O(mn)$, where m is the number of clauses, and n is the number of variables. Even though this algorithm runs slower than the implication graph based algorithm, it is worth mentioning because it uses a completely different approach in determining satisfiability.

4.3 Horn SAT

As is the case for 2CNF formulas, the satisfiability problem for Horn formulas is also in polynomial-time. The definition of Horn SAT is stated below:

> Horn SAT: Given a CNF formula F, where each clause in F has at most 1 positive literal (F is Horn), does there exist a truth assignment to the variables in F such that F evaluates to *true*?

In this section, we will present two polynomial-time algorithms that solve Horn SAT. The first algorithm results from the observation that a fact free Horn formula is always satisfiable. Recall from Section 3.3 that a fact in a Horn CNF formula is an unit clause containing a positive literal. A fact free Horn CNF then will have at least one negative literal in every clause. Therefore such a formula can be satisfied by assigning every variable to *false*.

The first Horn SAT algorithm uses this property, and it tests the satisfiability of a Horn CNF by attempting to remove all positive unit clauses. The pseudo code for the algorithm follows:

```
Procedure Horn-SAT1(F)
if F contains no positive unit clause
   return satisfiable;
else if F contains □
```

```
   return unsatisfiable;
else
   Horn-SAT1(PropUnit(F)); (PropUnit defined in Section 4.2 )
```

Basically, the algorithm uses unit propagation to propagate the truth assignments of variables in unit clauses until there are either no more positive unit clauses left, in which case the formula is satisfiable, or there is a contradiction in the truth assignment, meaning that the formula is unsatisfiable. In the worst case scenario, this algorithm runs in $O(n^2)$, where n is the number of literal occurrences in the Horn formula.

The second Horn SAT algorithm was presented by Downling and Gallier in [9] (with a minor correction by Scutellà in [21]). The algorithm converts a Horn formula to a labelled graph, and formulates the satisfiability problem as a data flow problem. The pseudo code below shows the graph building process:

Given a Horn CNF $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, with k variables
Construct $G(F)$, a labelled directed graph with $k+2$ nodes
(a node for each variable, plus 1 for **true** and 1 for **false**)
And add arcs as the following:
if C_i is a positive unit clause (q)
   add an arc from **true** to q labelled i;
if C_i is a negative clause, of the form $(\overline{p_1} \vee \ldots \vee \overline{p_x})$
   add x arcs from $p_1, \ldots, p_x$ to **false** labelled i;
if C_i is a definite Horn clause, of the form $(\overline{p_1} \vee \ldots \vee \overline{p_x} \vee q)$
   add x arcs from $p_1, \ldots, p_x$ to q labelled i;

The satisfiability problem of the original Horn formula becomes a data flow problem of the associated graph in which sets of paths called *pebblings* are sought. A pebbling is defined as follows:

Definition 4.6 *Let $G = (V, E)$ be an edge-labelled directed graph. There is a* pebbling *of a node $q \in V$ from a set of nodes $X \subseteq V$ if either $q \in X$, or for some edge label i (corresponding to the ith clause in the original Horn formula), $P_1, \ldots, P_q$ are the source nodes of all incoming edges to q labelled i, and there are pebblings from X to $P_1, \ldots, P_q$.*

A pebbling from a set of nodes X to a node q represents the implication in the Horn function $\bigwedge_{x \in X} x \rightarrow q$. To determine whether a Horn CNF formula

is satisfiable, we trace in the graph to see whether a pebbling exists that leads from the node *true* to the node *false*. This is stated in the following theorem:

Theorem 4.7 *(Theorem 3 of [9]) Let $G_A = (V, E)$ be the edge-labelled directed graph corresponding to a Horn formula A. If there is no pebbling of* false *from* {true} *then A is satisfiable.*

Dowling and Gallier provide two algorithms that use this theorem to determine the satisfiability of Horn formulas [9]. The first one starts at the *true* node, and explores the arcs in a breadth-first fashion. It maintains a queue of clauses, which initially is filled with only positive unit clauses of the Horn CNF (since they are the only ones that result in arcs from the *true* node to variables nodes). As each of these clauses are examined, additional clauses are added if they are reachable from current clauses, i.e. a pebbling from *true* to each of the variable nodes exists. If the *false* node turns out to be reachable in the breadth-first search for pebblings, then the Horn CNF is unsatisfiable.

Since each clause is entered into the queue at most once, and for each clause the algorithm examines and marks off some negative literals in order to determine whether a new clause should be added, the algorithm runs in time proportional to the number of occurrences of negative literals in the original Horn CNF, which is linear to the total number of literal occurrences.

The second pebbling searching algorithm reverses the direction of pebbling search, and begins at the *false* node. Recall that the original digraph was constructed so that if the starting nodes in a pebbling are assigned *true*, then the destination node must also be assigned *true*. Therefore in this bottom-up approach, the algorithm starts from the destination node *false*, and searches recursively in a depth-first manner back up the arcs to determine whether there is a pebbling that began at the *true* node. In order to facilitate this, the direction of the arcs in the implication graph of a Horn formula are reversed, and the *false* node is treated as the root node of the reverse-pebbling.

In this algorithm, for each clause of the Horn formula, each literal is visited once, and each of their truth values is computed exactly once. Therefore the running time of the algorithm is linear in the number of literal occurrences again.

5 Boolean formula minimization

The minimization problem for general Boolean formulas is a computationally hard problem. Simply put, the minimization problem for Boolean formulas is the decision problem of determining whether a given Boolean formula is of a certain minimal size in the number of literals. More specifically, this problem has been stated in two ways:

Minimum Equivalent Expression (MEE) [11]: Given a Boolean formula F and a nonnegative integer k, is it true that there exists a Boolean formula of size at most k that is equivalent to F?

Minimal [17]: Given a formula F, is it true that there does not exist a formula equivalent to F that is of smaller size than F?

5.1 The complexity of MEE

The minimization problem MEE has a trivial time complexity lower bound of coNP, and a trivial upper bound of Σ_2^p [17], meaning that it can be solved by a NP machine with a NP oracle. The coNP lower bound is derived as follows. It is known that the satisfiability problem for general Boolean formulas (SAT) is NP-complete. It follows then that the tautology problem, that is, whether a given Boolean formula evaluates to *true* for all truth assignments, is coNP-complete, meaning that the complement of the problem is NP-complete. Tautology can be formulated into an instance of the MEE problem, because a Boolean formula that is a tautology is equivalent to *true*, which is a Boolean formula of 0 literals. Therefore, to determine whether a Boolean formula F is a tautology, we can simply test to see whether the given formula has a equivalent representation of size 0, which in essence is an instance of MEE. However, a Boolean formula that always evaluates to *false*, a non-tautology, also has an equivalent reprensentation of size 0. Therefore, to include only formulas that are a tautology, we construct an instance of the MEE problem with the formula $F \vee p$, and $k = 0$, where p is any arbitrary variable that does not occur in F. This way, if F is a tautology, then $F \vee p$ also is a tautology. If F is a non-tautology, $F \vee p$ can still evaluate to *true* when p is assigned to *true*, hence, $F \vee p$ no longer has a equivalent representation of size 0. By the above, the coNP-hard lower bound of the minimization problem is established.

The upper bound of Σ_2^p comes from the fact that the problem can be solved by a NP machine with a NP oracle. This machine would guess a minimal representation non-deterministically, and ask the non-deterministic oracle to verify that it is equivalent to the original formula.

Recently, the lower bound for MEE has been raised to many-one hard for parallel access to NP [Hemaspaandra and Wechsung 1997]. In fact, MEE restricted to DNF formulas was shown to be complete for Σ_2^p by Umans in [23], and is Σ_2^p-hard to approximate within an n^ϵ factor [24].

5.2 The complexity of Minimal

Less is known about he complexity of Minimal. It was shown in [17] that the complement of Minimal, $\overline{\text{Minimal}}$ has an upper bound of Σ_2^p, and no better upper bound is known. In [15], it was shown that Minimal has a lower bound of coNP-hardness. In this paper, we do not discuss the minimization problem in the form of Minimal.

6 Minimization of Horn formulas

Because of the extensive use of Horn formulas in logic programming and relational databases, the minimization of Horn formulas has been the subject of much research. Finding the most compact representation of a Horn formula reduces the size of a knowledge base, and therefore can speed up operations in answering queries.

Different measurements for the size of the formula can yield different complexity results for the minimization problem. The most common measurements of formula size are the number of literal occurrences and the number of clauses, which we will call Horn Literal Minimization and Horn Term Minimization respectively. We state these two problems as follows:

Definition 6.1 Horn literal minimization*: Given a Horn CNF formula F and a positive integer k, is there a CNF equivalent to F that has at most k literals occurrences?*

Definition 6.2 Horn term minimization*: Given a Horn CNF formula F and a positive integer k, is there a CNF equivalent to F that has at most k terms?*

Using these two measurements, Horn minimization is NP-complete [13] [4]. In the following subsections, we will first show the NP upper bound of these two Horn minimization problems, then we will see two different problems that are known to be NP-complete reduced to the problem of Horn literal minimization and Horn term minimization.

6.1 Horn Literal Minimization and Horn Term Minimization are in NP

As presented in Section 4 of [13] and Lemma 3.1 of [4], both Horn Literal Minimization and Horn Term Minimization problems are in NP. For an instance of either of the Horn minimization problems, a certificate would be a formula F' that is claimed to be of size $\leq k$ and is equivalent to the original formula F. Given such a formula F', we verify the certificate by first counting the number of literals or the number of clauses (depending on the metric for size) in F' and see that it is less than or equal to k. Then we verify that the formula F' is indeed equivalent to F by checking that $F \rightarrow F'$ and $F' \rightarrow F$. To verify that F implies F', we verify that every clause in F' is implied by F. Given a clause C and a Horn formula F, this implication test assigns truth values to variables in C so that C is false, and then tests to see whether F is satisfiable with the partial truth assignment [13]. The clause C is implied by the CNF F if and only if F is unsatisfiable with this partial truth assignment. To verify that F' implies F, we simply do the reverse.

6.2 Reduction from set-cover

In [13], the NP-hardness of literal minimization for Horn CNF formulas is proven via a reduction from the known NP-complete problem of set-covering. In this section we present Hammer and Kogan's proof and reduction. We will first give the problem statement of minimum set covering. The reduction follows.

Minimum Set Covering: Given a collection $C = \{C_1, C_2, \ldots, C_m\}$ of subsets of a finite set $S = \{x_1, x_2, \ldots, x_n\}$, and a positive integer g, does C contain a cover for S of size at most g? That is, is there a sub-collection $C' \subseteq C$ with $|C'| \leq g$ such that every element of S belongs to at least one member of C'?

The set covering problem is formulated into a matrix in [13], which is uncommon. In their representation, each row of the matrix represents an element in the set, and each column of the matrix represents a subset of the elements. Here the reduction will be shown to work with the standard formulation of the set cover problem.

The reduction in [13] states that an instance of the Minimum Set Covering problem, with a collection of subsets C and a positive integer g can be reduced to an instance of Horn Literal Minimization with a Horn CNF F and a positive integer k like the following. First, a Horn CNF F is constructed from the collection of subsets in the set cover problem. We introduce a variable $y_j, j \in \{1, \ldots, m\}$ for each $C_j \in C$.

$$F = \left(\bigwedge_{j=1}^{m} \left(\bigwedge_{x_i \in C_j} (x_i \vee \overline{y_j}) \right) \right) \left(\bigwedge_{j=1}^{m} \left(y_j \vee \bigvee_{x_i \in C_j} \overline{x_i} \right) \right) \left(\bigvee_{i=1}^{n} \overline{x_i} \right)$$

For the sake of convenience, we shall use the following shorthand:

$$F_1 = \left(\bigwedge_{j=1}^{m} \left(\bigwedge_{x_i \in C_j} (x_i \vee \overline{y_j}) \right) \right)$$

$$F_2 = \left(\bigwedge_{j=1}^{m} \left(y_j \vee \bigvee_{x_i \in C_j} \overline{x_i} \right) \right)$$

$$F_3 = \left(\bigvee_{i=1}^{n} \overline{x_i} \right)$$

Then the positive integer k for the Horn Literal Minimization instance is derived from g of the Minimum Set Covering problem as follows:

$$k = m + 3 \sum_{j=1}^{m} |C_j| + g$$

Clearly, given an instance of the minimal set cover problem, the associated CNF F can be constructed in polynomial-time. Likewise, the positive integer k can be calculated from g in polynomial-time. Therefore, to prove the reduction, we only need to show that finding a CNF that equivalent to F with k literal occurrences is equivalent to finding a set cover of size g for the original instance of the set cover problem.

Let us first examine the reason for computing $k = m+3\sum_{j=1}^{m}|C_j|+g$. We see that F_1 has one clause for each element in each subset, and two literals per clause. Therefore the number of literals in the clauses that make up F_1 is $2\sum_{j=1}^{m}|C_j|$. F_2 has m clauses, one for each subset, and each clause corresponding to subset C_j contains $1+|C_j|$ literals. Therefore the number of literals in F_2 is $m+\sum_{j=1}^{m}|C_j|$, and the total number of literals in the first two components of F is $m+3\sum_{j=1}^{m}|C_j|$.

As will be shown later, the literal minimization of F involves only minimizing the number of literals in F_3, the negative clause in F. It turns out that the literal minimized formula F' will retain the first two components of F, and the third component is replaced by a clause that corresponds directly to a set of subsets that form a covering. In other words, the literal minimized formula is the Horn CNF $F' = F_1 \wedge F_2 \wedge F_3'$. This new clause F_3' has g literals, one for each subset in the covering. Therefore, a covering of size at most k' translates to a formula of at most $k = m+3\sum_{j=1}^{m}|C_j|+g$ literals, and vice versa.

Before going into further detail with the construction of the reduction, we will present some necessary tools from [13]. First, referring to their earlier paper [12], where it was shown that an arbitrary Horn function can be canonically decomposed into a negative part and a definite Horn part, Hammer and Kogan stated the following theorem in [13]:

Theorem 6.3 *(Theorem 2.5 in [13]) A conjunctive normal form F is an irredundant and prime representation of a Horn function f if and only if*

$$F = F(h(f)) \wedge F_N$$

where $F(h(f))$ is an irredundant prime CNF of the definite Horn component $h(f)$, and where F_N is a negative restriction of f.

For the purpose of our discussion, we only need to know that the conjunction of negative clauses in any prime and irredundant Horn CNF make up a negative restriction of the underlying function f (shown in [12]), and that all negative restrictions of a function has the same number of clauses. The core concept from Theorem 6.3 that will be needed for the reduction is that given a prime and irredundant Horn CNF $F = F_H \wedge F_N$ representing the Horn function f, where F_H is the conjunction of definite Horn clauses, and F_N is the conjunction of negative clauses (a negative restriction of f), we can construct a Horn formula $F' = F_H \wedge F_N'$ equivalent to F as long as

F'_N is a negative restriction of f. Then F_N is an implicate of $F_H \wedge F'_N$ and F'_N is an implicate of $F_H \wedge F_N$.

As another necessary tool, Hammer and Kogan defined a forward chaining procedure, which takes a subset of variables S as input, and returns a superset R of the input set. Details of the procedure is stated below.

Definition 6.4 *Forward chaining procedure: Given a subset of variables S, initialize $R = S$. Then at each step find a definite Horn clause C in which all the variables associated with the negative literals in C are in R, and that the variable associated with the positive literal is not in R. If such a clause is found, the variable associated with the positive literal is added to R. This is repeated as many times as possible, until no new variables are added to R.*

Recall that a definite Horn clause $(p \vee \overline{q_1} \vee \ldots \vee \overline{q_n})$ corresponds with the rule $p \leftarrow q_1 \wedge \ldots \wedge q_n$ in logic programming. The inclusion of p when $q_1, \ldots, q_n$ are all in R is thus called "firing the rule" $p \leftarrow q_1 \wedge \ldots \wedge q_n$. As it pertains to Horn formulas, we can see R as a subset of variables that are already assigned *true*. Then to satisfy the formula, whenever the variables associated with the negative literals in a definite Horn clause are in R, we must also assign the variable associated with the positive literal *true*. Also note that this forward chaining procedure is based on the same principle as following pebblings in Dowling and Gallier's paper regarding Horn SAT [9].

Using the forward chaining procedure, Hammer and Kogan established two lemmas that are crucial to the reduction. We will state them verbatim here. Please refer to [13] for further proofs.

Lemma 6.5 *(Lemma 3.1 in [13]) A definite Horn clause $C = x' \vee \bigvee_{x \in S} \overline{x}$ is an implicate of a definite Horn function h given by a Horn CNF F iff the forward chaining procedure starting with the set S includes eventually the variable x' into the set R.*

Lemma 6.6 *(Lemma 3.2 in [13]) A negative clause $C_1 = \bigvee_{x \in S_1} \overline{x}$ is an implicate of a Horn function $f = F \wedge C_2$ where F is a definite Horn CNF and $C_2 = \bigvee_{x \in S_2} \overline{x}$ is a negative clause, iff the forward chaining procedure for F starting with the set S_1 includes eventually every variable $x \in S_2$ into the set R.*

Armed with Theorem 6.3 and Lemmas 6.5 and 6.6, we are now ready to discuss the details of the reduction. As stated earlier, the reduction formulates a problem of finding a set covering from a collection of subsets into the

problem of minimizing the number of literal occurrences in a Horn formula F. To prove the reduction, it must be shown that a solution for a set cover problem can be translated into a solution for the corresponding Horn literal minimization problem. Likewise, a solution for a Horn literal minimization problem must be able to be translated into a solution for the corresponding set cover problem.

In the first direction, suppose that the collection of subsets C over the elements S have a covering $C' \subseteq C$ such that $|C'| \leq g$. Then using this covering C', the following formula can be constructed:

$$F' = \left(\bigwedge_{j=1}^{m} \left(\bigwedge_{x_i \in C_j} (x_i \vee \overline{y_j})\right)\right) \left(\bigwedge_{j=1}^{m} \left(y_j \vee \bigvee_{x_i \in C_j} \overline{x_i}\right)\right) \left(\bigvee_{C_j \in C'} \overline{y_j}\right)$$

First observe that the first two components are identical to F_1 and F_2 of the original formula F, thus have a total of $\left(m + 3\sum_{j=1}^{m} |C_j|\right)$ literal occurrences. The total number of literal occurrences in F' then is $\left(m + 3\sum_{j=1}^{m} |C_j| + g\right)$, which is k for our Horn literal minimization problem. It follows then that if F and F' are indeed equivalent, with the number of literals in F' being k, F' will be a solution to the Horn literal minimization problem corresponding to this set cover problem.

Here we realize the reason behind the construction of the structure of F. To show that $F \equiv F'$, we need to show that $F_3' = \left(\bigvee_{C_j \in C'} \overline{y_j}\right)$ is an implicate of $F_1 \wedge F_2 \wedge (\bigvee_{i=1}^{n} \overline{x_i})$, and that $F_3 = (\bigvee_{i=1}^{n} \overline{x_i})$ is an implicate of $F_1 \wedge F_2 \wedge \left(\bigvee_{C_j \in C'} \overline{y_j}\right)$. This is accomplished by Lemma 6.6 and the structure of F_1 and F_2.

First we show that F_3' is an implicate of $F_1 \wedge F_2 \wedge F_3$. Because of the structure of F_1 and according to Lemma 6.5, the forward chaining procedure starting with the variable set $\{y_j\}$ will eventually include every $\{x_i | x_i \in C_j\}$, corresponding to the elements in the subset C_j. Since M is a covering of the subsets, that means every element x_i is in at least one subset $C_j \in C'$. Then the forward chaining procedure starting with the set $\{y_j | C_j \in C'\}$ (from the clause F_3') will eventually include every x_i, thus every literal in the clause F_3. By Lemma 6.6, this proves that F_3' is an implicate of F.

Now we show that F_3 is an implicate of $F_1 \wedge F_2 \wedge F_3'$. By Lemma 6.6, if F_3 is an implicate of F', the forward chaining procedure starting with the set of all x_i (from the clause F_3), representing all elements, should eventually include all $\{y_j | C_j \in C'\}$ (from the clause F_3') into the set. Here we see the

reason behind the structure of F_2, where there is a clause for each subset C_j, in the form of $\left(y_j \vee \bigvee_{x_i \in C_j} \overline{x_i}\right)$. Since every x_i is in the initial set when the forward chaining procedure begins, all y_j will be included in the set at the end. It's obvious then that the variables $\{y_j | C_j \in C'\}$ representing the covering M, and more importantly, from the clause F_3', will all be included into the set. Therefore, F_3 is shown to be an implicate of F'.

Now we have shown how a solution to a set cover problem can be transformed into a solution to the corresponding Horn literal minimization problem, we will proceed to show the reverse.

First, Hammer and Kogan proved that the definite Horn part of F, namely $F_1 \wedge F_2$, is prime and irredundant. Then based on Theorem 6.3, $F_1 \wedge F_2$ will be a part of every prime and irredundant formula that is equivalent to F. Now suppose F can be represented by an equivalent prime and irredundant Horn formula F', where the number of literal occurrences in F' is $\leq k$. Then F' is a solution to the Horn literal minimization problem. It follows from Theorem 6.3 that F' will be in the form $F_1 \wedge F_2 \wedge F_3'$, where F_3' is a negative clause, in the form of $\left((\bigvee_{i \in P} \overline{x_i}) \vee (\bigvee_{j \in Q} \overline{y_j})\right)$. The number of literals in F_3' is at most $k - \left(m + 3\sum_{j=1}^{m} |C_j|\right)$, which is g.

Because F' is equivalent to F, F_3' is an implicate of F. Then by Lemma 6.6, the forward chaining procedure starting with the set $\{x_i | i \in P\} \bigcup \{y_j | j \in Q\}$ will eventually include all the variables in F_3 into the set, thus every x_i that were not in the set to begin with (those $\notin P$) will be included. The structure of F_1 asserts that this inclusion will take place if and only if all of the variables x_i that are added to the set by the forward chaining procedure belong to some subset $\{y_j | j \in Q\}$. Thus the collection of the subsets Q is a covering for all elements that are not in P. To achieve a full covering of all the element, for the elements that are in P, we can simply choose any arbitrary subset that contains it. Then the number of subsets that form this covering is $|P| + |Q|$, which is the number of literals in F_3'. And since the number of literals in F_3' is $\leq g$, we have a solution to the corresponding set cover problem.

The full proof can be found in [13].

6.3 Reduction from Hamiltonian-path

In [4], Boros and Čepek proved the NP-completeness of the Horn term minimization problem via a reduction from the problem of finding a Hamiltonian

path in an undirected cubic graph. Their result was actually stronger in that it proved that Horn term minimization is NP-complete even if the formula was restricted to pure Horn and that each clause had at most 3 literals. This is stated in their Corollary 2.1:

Theorem 6.7 *Corollary 2.1 of [4]: The problem of Horn term minimization remains NP-complete even if the input is restricted to the class of cubic pure Horn formulas.*

In their proof, Boros and Čepek looked at term minimization for Horn DNF formulas, we will adapt their proof to work for term minimization for Horn CNF formulas, as the two are computationally equivalent. We will also see that the same reduction will prove the NP-completeness of literal minimization of Horn formulas. This is stated in the following theorem:

Theorem 6.8 *The problem of Horn literal minimization is NP-complete even if the input is restricted to the class of cubic pure Horn formulas.*

6.3.1 Horn Term Minimization

The problem of Horn term minimization was stated in Section 6. The Hamiltonian Path problem, according to Garey and Johnson [11] is stated this way:

Hamiltonian Path:

Instance: Graph $G = (V, E)$, $V = \{x_1, \ldots, x_n\}$.

Question: Does G contain a Hamiltonian path, that is, an ordering of the vertices of G, $[x_1, x_2, \ldots, x_n]$ where $n = |V|$ and $(x_i, x_{i+1}) \in E$ for all $i, 1 \leq i < n$?

As stated earlier, Boros and Čepek's reduction utilizes Horn DNF formulas. We will transform their results into Horn CNF formulas. From an instance of the Hamiltonian Path problem, given a cubic graph G (a graph is cubic if every vertex is incident with exactly 3 edges), that has n vertices and m edges, we construct the following Horn CNF formula $F(G)$:

$$F(G) = \left(\bigwedge_{x_i \in V} (\overline{v} \vee x_i) \right) \wedge \left(\bigwedge_{(x_i, x_j) \in E} (\overline{x_i} \vee \overline{x_j} \vee v) \right)$$

The reduction is stated in the following theorem:

Theorem 6.9 *Theorem 2 of [4]: Let $G = (V, E)$ be a cubic graph. The associated pure Horn function $F(G)$ has a CNF representation of at most $m + 2$ terms if and only if G has a Hamiltonian path, i.e. a simple path that visits every vertex exactly once.*

In the construction of the formula $F(G)$, each vertex in G is identified with a variable x_i, and an additional Boolean variable v is added to the formula. Boros and Čepek further showed that a clause is a prime implicate of the function defined by $F(G)$ if and only if it belongs in one of the three categories listed below:

1. $(\overline{v} \vee x_k)$ for some $x_k \in V$.
2. $(\overline{x_i} \vee \overline{x_j} \vee x_k)$ for some $(x_i, x_j) \in E$ such that $k \notin \{i, j\}$.
3. $(\overline{x_i} \vee \overline{x_j} \vee v)$ for some $(x_i, x_j) \in E$.

The prime implicates in category 1 are simply clauses from the first component of $F(G)$, and the prime implicates in category 3 are clauses from the second component of $F(G)$. The prime implicates in category 2 are those that can be obtained through resolution on the variable v, of an implicate from category 1 and an implicate from category 3.

In the first direction of the reduction, suppose that G has a Hamiltonian Path $P = [x_i, x_{i+1}, \ldots, x_n]$. The same path can also be represented as a sequence of edges $P = [e_1, e_2, \ldots, e_{n-1}]$, where $e_i = (x_i, x_{i+1})$ for $1 \leq i \leq n - 1$. Then it must be shown that a Horn CNF equivalent to $F(G)$, and of at most $m+2$ clauses can be constructed from P. The CNF version of Boros and Čepek's formula is:

$$F' = (\overline{v} \vee x_1) \wedge (\overline{v} \vee x_2) \wedge (\bigwedge_{i=1}^{n-2} \overline{x_i} \vee \overline{x_{i+1}} \vee x_{i+2}) \wedge$$

$$(\overline{x_{n-1}} \vee \overline{x_n} \vee v) \wedge (\bigwedge_{(x_i, x_{i+1}) \in E \setminus P} \overline{x_i} \vee \overline{x_{i+1}} \vee v)$$

We see that F' has exactly $m + 2$ terms (the first two plus one for every edge in G), and that every clause in F' is a prime implicate of $F(G)$ because each falls in one of the three categories given above. To see that F' is equivalent to $F(G)$, we need to show that every clause in $F(G)$ can be derived from clauses in F', in other words, every clause in $F(G)$ is an implicate of F'. This

is indeed the case. First, every implicate $(\overline{v} \vee x_i)$ for $x_i \in V$ can be derived from resolution of the first two clauses with clauses in the third component. For instance, $(\overline{v} \vee x_3)$ is derived from resolving $(\overline{v} \vee x_1)$ and $(\overline{v} \vee x_2)$ with $(\overline{x_1} \vee \overline{x_2} \vee x_3)$, and $(\overline{v} \vee x_4)$ is derived from resolving $(\overline{v} \vee x_2)$ and $(\overline{v} \vee x_3)$ with $(\overline{x_2} \vee \overline{x_3} \vee x_4)$, and so on. Secondly, the implicates $(\overline{x_i} \vee \overline{x_{i+1}} \vee v)$ for $\{x_i, x_{i+1}\} \in P$ can be derived from resolving the forth term $(\overline{x_{n-1}} \vee \overline{x_n} \vee v)$ with terms from the third component.

Therefore, given a Hamiltonian path P of G, a CNF Horn formula F' with exactly $m+2$ terms can be constructed to represent the same function as $F(G)$. Hence one direction of the reduction is proven.

For the other direction of the reduction, suppose $F(G)$ has a CNF Horn representation F_0 containing at most $m+2$ terms. It must be shown that a Hamiltonian path of G can be constructed from the formula. This is indeed the case. Boros and Čepek show that through a series of modifications (modifications 0,1,2,3 in [4]) of F_0, an equivalent formula F_0' with exactly $m+2$ terms can be derived. Moreover, F_0' contains a set of clauses $(\overline{x_j} \vee \overline{x_{j+1}} \vee x_{j+2})$ for $j = 1, \ldots, \ell-2$, and $(\overline{x_{\ell-1}} \vee \overline{x_\ell} \vee v)$, from which a path in G can be constructed using edges $\{\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_{\ell-1}, x_\ell\}\}$.

Boros and Čepek further show that iterative modifications to the formula F_0', each time removing some terms and replacing some terms, will produce an equivalent formula with the same number of terms, and will extend the path that can be constructed until the path includes every vertex in G. Because all these modifications to the formula are polynomial-time, any CNF Horn formula equivalent to $F(G)$ containing at most $m+2$ terms can be translated to a Hamiltonian path of the associated graph G in polynomial-time. Thus the second direction of the reduction holds.

Details of the proof can be found in [4].

6.3.2 Horn Literal Minimization

We will now modify Boros and Čepek's reduction slightly to show that Horn literal minimization, as defined in Section 6 is also NP-complete. Just as we did for Horn term minimization, from an instance of the Hamiltonian Path problem, given a cubic graph G that has n vertices and m edges, we construct the following Horn CNF formula $F(G)$:

$$F(G) = \left(\bigwedge_{x_i \in V} (\overline{v} \vee x_i) \right) \wedge \left(\bigwedge_{(x_i, x_j) \in E} (\overline{x_i} \vee \overline{x_j} \vee v) \right)$$

To prove Theorem 6.8, by the end of this section, we will show that the following theorem is true:

Theorem 6.10 *Let $G = (V, E)$ be a cubic graph. The associated pure Horn function $F(G)$ has a CNF representation of at most $3m + 4$ literals if and only if G has a Hamiltonian path, i.e., a simple path that visits every vertex exactly once.*

In the first direction of the proof, suppose that G has a Hamiltonian Path $P = [x_i, x_{i+1}, \ldots, x_n]$. The same path can also be represented as a sequence of edges $P = [e_1, e_2, \ldots, e_{n-1}]$, where $e_i = (x_i, x_{i+1})$ for $1 \leq i \leq n-1$. We use the same formula as we did in the last section:

$$F' = (\overline{v} \vee x_1) \wedge (\overline{v} \vee x_2) \wedge (\bigwedge_{i=1}^{n-2} \overline{x_i} \vee \overline{x_{i+1}} \vee x_{i+2}) \wedge$$

$$(\overline{x_{n-1}} \vee \overline{x_n} \vee v) \wedge (\bigwedge_{(x_i, x_{i+1}) \in E \setminus P} \overline{x_i} \vee \overline{x_{i+1}} \vee v)$$

F' has exactly $3m+4$ literals (2 literals in each of the first two clauses, and 3 literals in each of the remaining clauses), and we have already shown that F' is equivalent to $F(G)$. Hence we are able generate a solution to the associated Horn literal minimization problem from a solution to the Hamiltonian path problem. This concludes the first direction of the proof.

For the other direction, we will use the exact same proof procedure as the one for Horn term minimization. Suppose $F(G)$ has a CNF Horn representation F_0 containing at most $3m + 4$ literals, it must be shown that a Hamiltonian path of G can be constructed from the formula. After applying modifications 0,1,2,3 of [4] to F_0, we will obtain a CNF formula equivalent to $F(G)$, with the following properties: it has exactly $m + 2$ terms; only two terms are in the form of $(\overline{v} \vee x_k)$ for some $x_k \in V$; and $n-2$ of the remaining m terms are in the form of $(\overline{x_i} \vee \overline{x_j} \vee x_k)$ for some $(x_i, x_j) \in E$ such that $k \notin \{i, j\}$, while the rest of the terms are in the form of $(\overline{x_i} \vee \overline{x_j} \vee v)$ for some $(x_i, x_j) \in E$. (page 11 of [4], based on Lemmas 4.5, 4.6a and 4.7 of [4]). Thus this Horn CNF has exactly $3m + 4$ literals.

Every subsequent modification to this formula (page 12 and 13 of [4]) does not alter the number of literals in the CNF. Therefore, after some iterations of these modification, a formula equivalent to $F(G)$ with exactly $3m + 4$ literal occurrences can be obtained from which a Hamiltonian path of G can be extracted.

6.4 Some properties of minimal formulas

Recall that an implicate of a CNF is prime if dropping any literal from it produces a Horn clause that is no longer an implicate, and that a CNF is irredundant if dropping any clause from it produces a formula that is not equivalent. The results in [12] imply the following lemmas:

Lemma 6.11 *The literal minimal CNF formula of a Boolean function must be both prime and irredundant.*

Proof This can be proved simply by contradiction. Suppose we have a literal minimal CNF formula F of some Boolean function f that is not prime. Then that means there exists at least one clause in F that is not prime. Replacing these clauses with their prime counterparts would produce a CNF that is equivalent to F but contains fewer literal occurrences, therefore F cannot be a literal minimal CNF of f.

Likewise, suppose we have a literal minimal CNF formula F of some Boolean function f that is not irredundant. Then that means there exists at least one clause in F that can be removed without altering the function represented. Removing these clauses would produce a CNF that is equivalent to F but contains fewer literal occurrences, therefore F cannot be a literal minimal CNF of f. ∎

Lemma 6.12 *The term minimal CNF formula of a Boolean function must be irredundant.*

Proof This is easily proved like the above, suppose we have a term minimal CNF formula F of some Boolean function f that is not irredundant. Then that means there exists at least one clause in F that can be removed without altering the function represented. Removing these clauses would produce a CNF that is equivalent to F but contains fewer clauses, therefore F cannot be a term minimal CNF of f. ∎

7 Minimization of Horn CNF of unit clauses

The simplest subclass of Horn CNF formulas are conjunctions of unit clauses, or clauses with a single literal. A Horn CNF of unit clauses really is just single level conjunction. It is trivial to see that such a Horn formula is both term minimal and literal minimal if and only if there are no redundant unit

clauses, that is to say, no literal occurs twice in the Horn conjunction. Each irredundant unit clause is necessary in the formula to represent the function, thus is an essential prime implicate of the function.

In order to minimize a Horn CNF of unit clauses, it is sufficient to go through the formula and eliminate redundant unit clauses. In the most naive approach, starting at the first clause, the algorithm would go through the rest of the formula clause by clause to eliminate repeated occurrences of the clause. If at any time the negation of the current clause is found, the formula is unsatisfiable, therefore can be replaced by *false*. Clearly, this algorithm would run in $O(n^2)$ where n is the number of clauses, which in this case equals the number of literals.

This elimination of redundant clauses can actually be done faster. The faster algorithm would first sort the clauses in alphabetical order, then simply go through the sorted formula just once, dropping any duplicate clauses. Because of the sorting, this algorithm runs in $O(n \lg n)$. The minimized formula has minimal number of clauses, and since every clause in the formula has exactly one literal, this minimization also minimizes the number of literal occurrences.

8 Minimization of purely negative Horn CNF

A purely negative Horn formula is a CNF that contains only negative clauses. Both literal minimization and term minimization of this subclass of Horn formulas are in P.

For term minimization, according to Corollary 5.4 in Hammer and Kogan's *Horn functions and their DNFs* [12], "all the irredundant prime DNFs of Horn functions contain the same number of positive terms." It follows then that all the irredundant prime CNFs of Horn functions contain the same number of negative terms. Therefore, a negative Horn formula is term minimized when it is reduced down to an irredundant equivalent.

As for literal minimization, we can see that a purely negative CNF has no implicates other than the ones that already exist in the CNF as clauses, since no new clauses can be derived using resolution. We then only need to make sure that no clause in the CNF is implied by another clause in the CNF. Therefore, given a purely negative Horn CNF, we can minimize the number of literal occurrences by reducing it to a prime and irredundant equivalent CNF.

Therefore both literal minimization and term minimization can be done using Hammer and Kogan's algorithm in [13], which first makes every clause a prime implicate, then purges redundant implicates. The algorithm returns irredundant CNF that is equivalent to the input CNF, and runs in $O(n^2)$ where n is the number of literals. [13]

9 Minimization of 2CNF

Another restricted subclass of Horn CNF formulas is 2-Horn CNF, in which every clause has at most 2 literals. As will be shown in Section 11, functions represented by 2-Horn CNF formulas are a subset of what [14] calls *quasi-acyclic*, and it was shown in [14] that literal minimization for quasi-acyclic Horn functions is in polynomial-time.

In this section, we will show that both term minimization and literal minimization for general 2CNF formulas are in P. We present here an algorithm that minimizes both the number of clauses and the number of literal occurrences in a 2CNF formula.

The minimization algorithm has the following steps.

1. Purge the formula of useless clauses
2. Check that the formula is satisfiable
3. Via resolution, find all unit implicates of the function and divide the resulting formula into two parts; a conjunction of unit implicates, and a conjunction of 2-literal clauses that do not have any unit implicates
4. Convert the conjunction of 2-literal clauses into an implication graph
5. Find the transitive reduction of the 2CNF implication graph
6. Convert the reduced graph back to a 2CNF formula, then conjunct it with the unit implicates

We will discuss each of the steps in detail.

Purging the formula of useless clauses

A clause of the form $(p \vee \overline{p})$ will always evaluate to *true* and thus provides no useful information about the function represented by the formula. Removing useless clauses from a formula will have no effect on the underlying function represented by the formula. To remove all useless clauses, we can simply go through the formula once and discard every clause that always evaluates to *true*. If the 2CNF formula contains any useless clauses, then this removal procedure reduces the number of terms (1 for each clause removed), as well as the number of literal occurrences (2 for each clause removed). If every clause in the 2CNF formula is a useless clause and this useless clause removal process returns an empty formula, then the formula is a tautology, and can be replaced by *true*. When a formula is replaced by *true*, the formula is already minimized, and the algorithm can stop here. On the other hand, if a 2CNF formula is not a tautology, then the algorithm continues with the following steps to minimize it. In Section 9, we will explain why this step is necessary.

Checking the satisfiability of the formula

An unsatisfiable formula is false for all truth assignments of the variables. Therefore any formula of an unsatisfiable Boolean function can be replaced with the minimal formula *false*, which contains no clauses and no literals. As shown in Section 4.2, we can check the satisfiability of a 2CNF formula in polynomial-time. Therefore in this step, we simply use a satisfiability check for 2CNF formulas, and if the formula is unsatisfiable, then we replace it with *false*.

If the formula is neither unsatisfiable nor a tautology, then we proceed to minimize it with the following steps.

Unit resolution, and dividing the formula into two parts

Unit resolution is a special type of resolution, in which one of the two resolving clauses is always a unit clause. Unit resolution has been used extensively in SAT solvers for 2CNF. In this step of the algorithm, we use unit resolution based on the Davis Putnam procedure, as discussed in Section 4.2, to reduce a 2CNF formula by uncovering all the unit implicates of the formula. This step is described in further detail below.

Given a 2CNF formula, we first use unit propagation to uncover more unit clauses from 2-literal clauses. Unit propagation works as follows. Every unit clause in the 2CNF formula has to evaluate to *true* in order for the formula to evaluate to *true*. Therefore we must assign truth assignments to the variables that occur in unit clauses so that each one will evaluate to *true*. For instance, if (u) is a clause in the 2CNF F, then u must be assigned *true*. Each time a variable occurring in an unit clause is assigned a truth value, we propagate the truth assignment to other instances of literals of the this variable. For instance, following the previous example, since u is assigned *true*, every clause in F in the form of $(u \vee v)$ can be discarded, because each already evaluates to *true*, and every clause in F in the form of $(\overline{u} \vee v)$ can be reduced to (v), a new unit clause. We continue this unit propagation process for each unit clause, including new unit clauses derived from unit propagation, every variable occurring in an unit clause is assigned a truth value.

Then for every variable v in F that is not yet assigned a truth value, we want to know whether any of them has to be assigned either *true* or *false* in order for the formula to be satisfiable. To examine a variable v, we test the satisfiability of $F \vee (v)$ and the satisfiability of $F \vee (\overline{v})$. We do this by first assigning v to *true* and testing the satisfiability of F, then assigning v to *false* and testing the satisfiability of F. If $F \vee (v)$ is unsatisfiable, i.e., if assigning v to *true* renders F unsatisfiable, then we add the unit clause $(\overline{v})$ to F. On the other hand, if $F \vee (\overline{v})$ is unsatisfiable, we add the unit clause (v) to F. Then we carry out unit propagation with this newly assigned value of v. If F is satisfiable regardless of the truth assignment of v, then we do not give it an assignment and move on to the next unexamined variable.

The following pseudo code presents the unit resolution algorithm more succinctly. Note that PropUnit has been slightly modified to account for the marking of variables, and the fact that the input 2CNF formula has been verified to be satisfiable.

```
Given a satisfiable 2CNF formula F:

Procedure UnitResolution(F)
F := PropUnit(F);
while there exists unmarked variables in F
   choose an unmarked variable x;
   P := PropUnit(F ∧ (x));
   N := PropUnit(F ∧ (¬x));
```

```
   if P is satisfiable
      if N is unsatisfiable
         F := P;
   else
      F := N;
   mark x;

Procedure PropUnit(F)
while F contains unit clauses of unmarked variables
   choose a unit clause U containing an unmarked variable
   for each clause C in F
      if C contains U
         F = F\{C};
      else if C contains ¬U
         F = F\{C} ∧ {C\¬U};
      return F;
```

When we are finished, we have found every variable in the function represented by F that has to be assigned either *true* or *false* for the function to evaluate to *true*. These variables are represented in the resulting CNF formula as unit implicates, a positive literal for each variable that has to be assigned *true*, and a negative literal for each variable that has to be assigned *false*. There may be still be clauses with 2 literals, but because of unit propagation, these 2-literal clauses will not have variables in common with the unit clauses. So we see that the resulting formula from unit resolution is a conjunction $F_1 \wedge F_2$, where F_1 is a conjunction of unit implicates, and F_2 is a conjunction of 2-literal clauses such that F_1 and F_2 are over disjoint sets of variables, and F_2 has no unit implicates. For our convenience, we will define the latter part as follows.

Definition 9.1 *Given a 2CNF formula F, F will be called a* pure 2CNF *if and only if every clause in F contains exactly 2 literals, and F does not have any unit implicates.*

Pertaining to minimization, the two parts of the formula have the following important property.

Lemma 9.2 *Given a CNF formula $F = F_1 \wedge F_2$, where F_1 is a conjunction of all unit implicates, and F_1 and F_2 are over disjoint sets of variables, computing a CNF formula F' such that $F \equiv F'$ and there exists no other CNF formula equivalent to F that has fewer clauses than F' (term-minimizing) or fewer number of literal occurrences than F' (literal-minimizing) amounts to term-minimizing and literal-minimizing F_2.*

Proof As shown in Section 7, every unit implicate of a Boolean function is essential. This means that every unit implicate will be present in any prime CNF formula of the same function. We know that the literal-minimal CNF formula must be prime and irredundant (Lemma 6.11), and that there exists a term-minimal CNF formula that is prime and irredundant. Therefore, every unit implicate will be a clause in every literal-minimal CNF formula representation, and every unit implicate will be a clause in a term-minimal CNF formula representation. F_1 then is a fixed part of a term-minimal and all literal-minimal CNF representation of the function represented by F.

Moreover, as a result of exhaustive unit propagation, F_1 and F_2 will be over disjoint sets of variables. Therefore the minimization of F_1 and F_2 can be done independent of each other, and $(F_1 \wedge F_2) \equiv (F_1 \wedge F_2')$ as long as $F_2 \equiv F_2'$. ∎

Converting the pure 2CNF part to a implication digraph

As shown in Section 4.2, to convert a 2CNF formula to a directed graph, Aspvall, Plass and Tarjan presented the following technique in [3]:

Given a 2CNF F,

1. For each variable p_i in F, add two vertices p_i and $\overline{p_i}$ to the graph. p_i and $\overline{p_i}$ are considered complements of each other.

2. For each clause $(u \vee v)$ in F, add an arc from $\overline{u}$ to v and an arc from $\overline{v}$ to u to the graph.

The arcs are added in such a way because each clause $(u \vee v)$ can be interpreted as two logical implications, $\overline{u} \rightarrow v$ and $\overline{v} \rightarrow u$. We represent these implications in the implication graph by the arcs $\overline{u} \rightarrow v$ and $\overline{v} \rightarrow u$ respectively. We will refer to these two arcs as mirror arcs of each other.

Definition 9.3 *In a 2CNF implication digraph, the arcs $\overline{u} \rightarrow v$ and $\overline{v} \rightarrow u$, corresponding to the clause $(u \vee v)$ in the 2CNF formula are mirror arcs of each other.*

Since every arc in the implication digraph of a pure 2CNF has a mirror arc, similarly, every path from u to v will have a mirror path from $\overline{v}$ to $\overline{u}$. The arcs making up a path, $u \to x_1, x_1 \to x_2, \ldots, x_n \to v$ have their corresponding mirror arcs $\overline{x_1} \to \overline{u}, \overline{x_2} \to \overline{x_1}, \ldots, \overline{v} \to \overline{x_n}$, which in reverse make up the mirror path from $\overline{v}$ to $\overline{u}$. Thus we can define mirror paths in the context of 2CNF implication graphs as follows.

Definition 9.4 *In a 2CNF implication digraph, the paths* $u \to x_1 \to x_2 \to \ldots \to x_n \to v$ *and* $\overline{v} \to \overline{x_n} \to \ldots \to \overline{x_2} \to \overline{x_1} \to \overline{u}$ *are mirror paths of each other.*

The implication graph can be used to test the satisfiability of the formula, as stated in Theorem 4.5.

Since pure 2CNF formulas are a subset of general 2CNF formulas, Theorem **??** holds true for pure 2CNF formulas. The satisfiability testing in Theorem **??** is unnecessary for our purposes since the original 2CNF formula has been verified to be satisfiable in an earlier step of the algorithm. However, Theorem **??** is useful to show that no vertex and its complement in the implication digraph will be in the same strongly connected component.

Essentially, the implication digraph of a 2CNF formula is just another way to represent the underlying Boolean function, embodying all the implications in the formula in the arcs and paths. The following lemma follows almost immediately:

Lemma 9.5 *Given a pure 2CNF* F*, a clause* $C = (a \vee b)$ *is an implicate of* F*, in other words,* $F \to C$*, if and only if a path from* $\overline{a}$ *to* b *and a path from* $\overline{b}$ *to* a *exist in the implication graph* $G(F)$ *of* F*.*

Proof First we prove the right to left direction. F implying $(a \vee b)$ can only be true if either F contains the clause $(a \vee b)$, or F contains a set of clauses that imply $(a \vee b)$.

In the first case, the arcs $\overline{a} \to b$ and $\overline{b} \to a$ are added to the implication graph $G(F)$ in the construction. Therefore a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in the implication graph of F simply by ways of these two arcs.

In the second case, for a subset of clauses in F to imply $(a \vee b)$, both a and b must be in at least one of the clauses. Furthermore, these clauses must be able to be arranged in such an order to form a interconnected chain starting with a and ending with b, i.e. $(a \vee p_1) \wedge (\overline{p_1} \vee p_2) \wedge (\overline{p_2} \vee p_3) \wedge \ldots \wedge (\overline{p_{n-1}} \vee p_n) \wedge (\overline{p_n} \vee b)$.

The first condition is obvious, since a conjunction of 2-literal clauses cannot imply $(a \vee b)$ if either a or b is not present in any of the clauses. The second condition is a necessity for resolution of the clauses to cause a domino effect, resulting in $(a \vee b)$.

These clauses will result in the following arcs in the construction of the implication graph: $\overline{a} \rightarrow p_1, p_1 \rightarrow p_2, p_2 \rightarrow p_3, \ldots, p_{n-1} \rightarrow p_n, p_n \rightarrow b$, forming a path from $\overline{a}$ to b, and $\overline{b} \rightarrow \overline{p_n}, \overline{p_n} \rightarrow \overline{p_{n-1}}, \ldots, \overline{p_3} \rightarrow \overline{p_2}, \overline{p_2} \rightarrow \overline{p_1}, \overline{p_1} \rightarrow a$, forming a path from $\overline{a}$ to b.

Now we prove the left to right direction. A path from $\overline{a}$ to b in the implication graph is made up of arcs leading from $\overline{a}$ to b, for example $\overline{a} \rightarrow p_1, p_1 \rightarrow p_2, p_2 \rightarrow p_3, \ldots, p_{n-1} \rightarrow p_n, p_n \rightarrow b$. By the construction of the implication graph, each arc $u \rightarrow v$ (from vertex u to vertex v) represents the logical implication $u \rightarrow v$ (from variable u to variable v). Thus the path $\overline{a}$ to b in the implication graph implies the logical implication $\overline{a} \rightarrow b$. Likewise, A path from $\overline{b}$ to a implies the logical implication $\overline{b} \rightarrow a$. Therefore, if a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in the implication graph, then both logical implications $\overline{a} \rightarrow b$ and $\overline{b} \rightarrow a$ exist in the function. Since $(\overline{a} \rightarrow b) \wedge (\overline{b} \rightarrow a) \equiv (a \vee b)$, and since F represents the same function as the implication graph $G(F)$, F must imply $(a \vee b)$. ∎

Obtaining the transitive reduction of the implication graph

A transitive reduction G^t of a graph G is defined in [1] by the following two conditions:

1. there is a directed path from vertex u to vertex v in G^t if and only if there is a directed path from u to v in G, and

2. there is no graph with fewer arcs than G^t satisfying condition (1).

The first part of the definition states that all paths in a graph G will be preserved in its transitive reduction G^t. Since a path in the implication graph of a CNF formula represents an implication in the underlying function, the transitive reduction of said graph preserves all implications. Therefore, the following corollary can be made from Lemma 9.5.

Corollary 9.6 *Given a pure 2CNF formula F and its implication graph $G(F)$, $F \rightarrow (a \vee b)$ if and only if a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in the transitive reduction of $G(F)$.*

Using this corollary, we can state the following.

Lemma 9.7 *Let F_1 and F_2 be two pure 2CNF formulas over the same set of variables. $F_1 \equiv F_2$ if and only if the transitive reductions of the implication graphs of F_1 and F_2 coincide.*

Proof $(F_1 \equiv F_2)$ means that F_1 and F_2 represent the same Boolean function. Therefore F_1 and F_2 must imply exactly the same implications among the variables. In other words, $F_1 \rightarrow (a \vee b)$ if and only if $F_2 \rightarrow (a \vee b)$.

By Corollary 9.6, $F_1 \rightarrow (a \vee b)$ if and only if a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in a transitive reduction of the implication graph of F_1. Likewise, $F_2 \rightarrow (a \vee b)$ if and only if a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in a transitive reduction of the implication graph of F_2. Since $F_1 \rightarrow (a \vee b)$ if and only if $F_2 \rightarrow (a \vee b)$, the transitive reductions of the implication graphs $G(F_1)$ and $G(F_2)$ coincide, that is, a path from u to v exists in a transitive reduction of $G(F_1)$ if and only if a path from u to v exists in a transitive reduction of $G(F_2)$.

In the other direction, if the transitive reductions $G(F_1)^t$ and $G(F_2)^t$ of the implication graphs $G(F_1)$ and $G(F_2)$ coincide, then a path from any vertex a to any other vertex b exists in $G(F_1)^t$ if and only if a path from a to b exists in $G(F_2)^t$. From Corollary 9.6, a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in $G(F_1)^t$ if and only if $F_1 \rightarrow (a \vee b)$. Now if $G(F_1)^t$ and $G(F_2)^t$ coincide, a path from $\overline{a}$ to b and a path from $\overline{b}$ to a exist in $G(F_1)^t$, if and only if a path from $\overline{a}$ to b and a path from $\overline{b}$ to a also exist in $G(F_2)^t$, and subsequently, $F_1 \rightarrow (a \vee b)$ if and only if $F_2 \rightarrow (a \vee b)$. ∎

We have shown so far that a pure 2CNF formula F can be represented by the implication graph $G(F)$ of F, and in turn can be represented by a transitive reduction of $G(F)$. Because a transitive reduction of $G(F)$ preserves every implication implied by F, and only those implications implied by F, while minimizing the number of arcs, we can minimize a pure 2CNF formula by finding a transitive reduction of the associated implication graph. However, this is not a straight-forward process. As we show in the next section, extra caution must be taken when computing the transitive reduction in order to obtain a graph that can be converted back into a 2CNF formula.

To convert the transitive reduction to a 2CNF

There is a certain form that implication graphs must adhere to in order to be converted back to a pure 2CNF formula, namely, if there is an arc from u to

v, then there must be an arc from $\overline{v}$ to $\overline{u}$. We will call this the *2CNF form*. The 2CNF form must be preserved when we compute the transitive reduction in order for the resulting transitive reduction to be convertible back into a 2CNF formula.

We will now present a modified version of the algorithm presented in [1]. Just as Aho et al. investigated separately the transitive reduction of acyclic digraphs and digraphs that contain cycles, We will divide the following discussion into these two categories.

- The implication graph is acyclic:

  Aho et al. stated in their Theorem 1 in [1] that any finite acyclic directed graph has an unique transitive reduction. They further stated that the unique transitive reduction can be obtained by inspecting each of the arcs of the graph one by one in any order, and removing any arc that is redundant. A *redundant arc* here is defined as an arc that connects a vertex u to a vertex v when there exists at least one other directed path from u to v that does not include this arc.

  Since there is a unique transitive reduction for any digraph, if the transitive reduction of a 2CNF implication digraph always retains the 2CNF form, then this simple removal of redundant arcs will result in an implication graph that can be converted back to a minimized 2CNF formula. This is in fact the case, as the following lemmas show.

  Lemma 9.8 *In an acyclic implication graph of a pure 2CNF formula, an arc $u \rightarrow v$ is redundant if and only if the mirror arc $\overline{v} \rightarrow \overline{u}$ is redundant.*

  Proof If an arc $u \rightarrow v$ is determined to be redundant in G, then there exists an alternative path from u to v that does not include this arc $u \rightarrow v$. Since every arc in G has a mirror arc, and every path in G has a mirror path, this alternative path from u to v also has a corresponding mirror path from $\overline{v}$ to $\overline{u}$ that does not include the arc $\overline{v} \rightarrow \overline{u}$. This makes the arc $\overline{v} \rightarrow \overline{u}$ also redundant.

  The proof in the other direction is the same. If an arc $\overline{v} \rightarrow \overline{u}$ is redundant, then there exists an alternative path from $\overline{v}$ to $\overline{u}$ that does not include the arc $\overline{v} \rightarrow \overline{u}$. The mirror path of this alternative path is a path from u to v that does not contain the arc $u \rightarrow v$. Therefore, the arc $u \rightarrow v$ is redundant. ∎

The converse of Lemma 9.8 is the following corollary:

Corollary 9.9 *In an acyclic implication graph of a pure 2CNF formula, an arc $u \rightarrow v$ is irredundant if and only if the mirror arc $\overline{v} \rightarrow \overline{u}$ is irredundant.*

Lemma 9.10 *Given an acyclic implication digraph $G(F)$ of a pure 2CNF formula F, the transitive reduction $G(F)^t$ of $G(F)$ retains the 2CNF form, i.e., the arc $u \rightarrow v$ is in $G(F)^t$ if and only if $\overline{v} \rightarrow \overline{u}$ is in $G(F)^t$.*

Proof Suppose that in a implication graph G of a pure 2CNF formula, the arc $u \rightarrow v$ is found to be redundant, then by Lemma 9.8, $\overline{v} \rightarrow \overline{u}$ is also redundant. By Lemma 1 in [1], the transitive reduction is obtained by removal of redundant arcs in any order. This means that if an arc a is redundant, and a different redundant arc is removed, a is still redundant in the resulting graph. Thus the transitive reduction of G, $G^t = (G - \{u \rightarrow v\})^t = (G - \{\overline{v} \rightarrow \overline{u}\})^t = (G - \{u \rightarrow v\} - \{\overline{v} \rightarrow \overline{u}\})^t$. This means that both $u \rightarrow v$ and its mirror arc $\overline{v} \rightarrow \overline{u}$ will be removed in the process of computing the transitive reduction.

On the other hand, suppose the arc $u \rightarrow v$ is found to be irredundant, then its mirror arc $\overline{v} \rightarrow \overline{u}$ will also be irredundant. That means both arcs will be preserved in the transitive reduction.

Therefore, regardless of the order of removal, pairs of redundant arcs will be removed by the end, each arc with its mirror arc. And every irredundant arc preserved would also have its mirror arc preserved. Since the graph starts out in the 2CNF form, removing only arcs and their mirrors does not alter the 2CNF form. ∎

- the implication graph contains cycles:

  When finding the transitive reduction of digraphs with cycles, Aho et al. [1] (pp. 133-135) use the following 3 step algorithm:

  Given a directed graph G,

  1. Obtain the compacted acyclic graph (called equivalent acyclic graph in [1]) G_1 of G.

2. Find the unique transitive reduction G_2 of G_1.
3. Obtain a cyclic expansion G_3 of G_2.

Each step is discussed below in detail:

The compacted acyclic graph G_1 of G is obtained by replacing each maximally strongly connected component in G with a new vertex s_i, which we will call a *strongly connected vertex*. S_i will denote the set of vertices in G that are in the strongly connected component replaced by s_i. Each vertex in G that is not in a in a strongly connected component will be replaced by an unique strongly connected vertex s_j, where it is the only member in the set S_j. Arcs are then added to G_1 such that there is an arc from s_j to s_k if and only if there exists an arc in G from a vertex u to a vertex v such that $u \in S_j$ and $v \in S_k$.

Step 2 of the algorithm simply computes the transitive reduction G_2 by removing redundant edges from the acyclic directed graph G_1 in any order, as discussed in the previous subsection. Naturally, G_2 will be over the same set of vertices as G_1.

In step 3, a cyclic expansion G_3 of G_2 is constructed by replacing each strongly connected vertex in the following manner: If there are multiple vertices in S_i, then s_i in G_2 is replaced by a simple cycle through all the vertices in S_i. If there is only one vertex in S_i, then s_i in G_2 is replaced by the vertex that is in S_i. Finally, each arc in G_2 from s_m to s_n is replaced in G_3 by one single similarly directed arc between some pair of vertices u and v, such that $u \in S_m$ and $v \in S_n$. Aho et al. proved that the resulting cyclic expansion G_3 is a transitive reduction of the original directed graph G in their Theorem 2.

As noted earlier, in a implication graph G of a pure 2CNF formula, each arc has a mirror arc, and each path has a mirror path. As a result, each maximally strongly connected components in G also has a mirror strongly connected component. We will call the strongly connected vertices that replaced these strongly connected components in step 1 mirrors of each other, as well as single member strongly connected vertices that replace complementing vertices in G. We will denote mirror strongly connected vertices by S_i and $\overline{S_i}$. Similarly, in step 3, when a strongly connected vertex is replaced by a simple cycle, we will call the cycles that replace pairs of mirrored strongly connected vertices mirror cycles of each other.

Using the three-step algorithm of Aho et al. to reduce a 2CNF implication graph may alter the 2CNF form of the graph in two different ways. They both occur in step 3. We can see that each maximally strongly connected component in G is essentially reduced to a simple cycle in the transitive reduction G_3 by the end of the algorithm. The first problem occurs when a resulting simple cycle and its mirror cycle are not 'wired' in the correct order to maintain the 2CNF form. For instance, a cycle $a \to b, b \to c, c \to d, d \to a$, and $\overline{a} \to \overline{d}, \overline{d} \to \overline{b}, \overline{b} \to \overline{c}, \overline{c} \to \overline{a}$ are simple cycles over two mirror connected vertices sets. However, the arcs in one cycle are not mirror arcs the arcs in the other cycle, so the 2CNF form does not hold.

The second problem occurs when the simple cycles and their mirror cycles are in 2CNF form, but arcs connecting the cycles are not mirror arcs of each other. For instance, a cycle $a \to b$, $b \to c$, $c \to a$ can connect to another cycle $d \to e$, $e \to f$, $f \to d$ via the arc $a \to d$, while the two mirror cycles $\overline{a} \to \overline{c}, \overline{c} \to \overline{b}, \overline{b} \to \overline{a}$ and $\overline{d} \to \overline{f}, \overline{f} \to \overline{e}, \overline{e} \to \overline{d}$ connect via the arc $\overline{f} \to \overline{c}$. Obviously, the 2CNF form is not maintained in this example.

To avoid altering the 2CNF form while reducing the graph, we modify step 3 of Aho et al.'s algorithm to obtain a cyclic expansion as follows:

1. Replace each strongly connected vertex s_i where S_i contains more than one vertex, and the mirror strongly connected vertex of s_i with simple cycles that are mirrors or each other, that is, arcs in one cycle are mirror arcs of the other.
2. Replace each strongly connected vertex s_j where S_j contains a single vertex with the one vertex in it.
3. Replace arcs between strongly connected vertices in G_2 like the following: If there is an arc from s_i to s_j in G_2, replace it by one single similarly directed arc between some pair of vertices $u \to v$, such that $u \in S_i$ and $v \in S_j$. Then replace the arc from $\overline{s_j}$ to $\overline{s_i}$ by $\overline{v} \to \overline{u}$.

It is easy to see that the 2CNF form will remain intact in the final transitive reduction with this modification of the algorithm since every arc $u \to v$ will have its mirror arc $\overline{v} \to \overline{u}$ restored in the expansion, and thus every path from u to v will have its mirror path from $\overline{v}$ to $\overline{u}$ in

the resulting transitive reduction. Furthermore, this modification only restricts the way in which the cyclic expansion is constructed, therefore the results of Aho et al. still hold.

Since the 2CNF form can be preserved in both acyclic and cyclic implication digraphs, the resulting transitive reduction can be converted back into a 2CNF formula. Now we can state the following:

Theorem 9.11 *Given a satisfiable 2CNF F with no unit implicates and no useless clauses, let $G(F)$ be the implication graph of F, let $G(F)^t$ be the transitive reduction of $G(F)$, and let F' be the 2CNF that is converted from $G(F)^t$. Then $F' \equiv F$ and there is no equivalent 2CNF that contains fewer clauses or fewer literals than F'. In other words, F' is a term-minimized and literal-minimized representation of the function represented by F.*

Proof We have already established by Lemma 9.7 that F and F' are equivalent. Therefore we only need to prove that F' is a term minimized and literal minimized representation of the function.

Suppose for a contradiction that there exists another pure 2CNF formula F'' that is equivalent to F, and contains fewer clauses than F'. Let us further suppose that the number of clauses in F'' is m, and the number of clauses in F' is n. Then the number of arcs in the implication graph that can be constructed from F'' is $2m$, and the number of arcs in the implication graph of F' is $2n$, with $2m < 2n$. This is a contradiction to the definition of transitive reduction, hence we have proved that no other formula is equivalent to F and contains fewer clauses than F'.

The number of literal occurrences in a pure 2CNF is simply twice the number of clauses in the formula, because each clause has exactly 2 literals. Just as the above proof, suppose for a contradiction that there exists another pure 2CNF formula F'' that is equivalent to F, and contains fewer number of literal occurrences than F'. Let us further suppose that the number of literal occurrences in F'' is m, and the number of literal occurrences in F' is n. Then the number of clauses in F'' is $m/2$, and the number of clauses in F' is $n/2$. Therefore number of arcs in the implication graph that can be constructed from F'' is m, and the number of arcs in the implication graph of F' is n, with $m < n$. This is a contradiction to the definition of transitive reduction, hence we have proved that no other formula is equivalent to F and contains fewer number of literal occurrences than F'. ∎

The algorithm presented here returns a 2CNF formula as the minimized equivalent formula. We need to be certain that minimizing 2CNF formulas amounts to looking at only 2CNF formulas. In other words, we need to show that given a 2CNF formula to minimize, we would always be able to find a 2CNF formula that has no more clauses and literal occurrences than any other CNF formula representing the same function, regardless of the number of literals per clause.

Lemma 9.12 *Let F be a satisfiable 2CNF formula, if $F \equiv G$, where G is a term-minimal and literal-minimal CNF, then there exists a $\widehat{G}$ in 2CNF such that $F \equiv \widehat{G}$ and the number of terms in $\widehat{G}$ is equal to the number of terms in G, and the number of literal occurrences in widehat$G$ is equal to the number of literal occurrences in G.*

Proof As shown in Lemma 3.2, using resolution, all prime implicates will be found. It can easily be seen that starting with a 2CNF formula, where every clause has at most 2 literals, resolution of any two clauses produces a resolvent that contains at most two literals. For instance, $(a \vee b)$ resolved with $(\overline{a} \vee c)$ results in $(b \vee c)$.

Since the prime implicates are either unit implicates or 2-literal implicates, we need only to examine equivalent formulas in 2CNF to find a term-minimal and literal-minimal representation of the underlying function. In other words, for any satisfiable 2CNF formula, there exists a term-minimal and literal-minimal equivalent formula in 2CNF. ∎

The reason for removing useless clauses preprocessing

The first step of the minimization algorithm is to remove all clauses of the form $(u \vee \overline{u})$. This turns out to be a necessary step because clauses of this form are not always removed by the rest of the algorithm. Consider the 2CNF $F = (u \vee \overline{u}) \wedge (v \vee w)$. When constructing the implication graph $G(F)$ of F, the arcs $u \rightarrow u$ and $(\overline{u} \rightarrow \overline{u}$ are added for the clause $(u \vee \overline{u})$, and the arcs $\overline{v} \rightarrow w$ and $(\overline{w} \rightarrow v$ are added for the clause $(v \vee w)$. $G(F)$ is already a the unique transitive reduction of itself and the self-loops in the graph are not removed. Therefore the algorithm returns the same formula $(u \vee \overline{u}) \wedge (v \vee w)$ when $(v \vee w)$ is shorter and equivalent. Therefore, the removal of useless clauses is an integral part of the minimization algorithm.

The reason for unit resolution preprocessing

Likewise, unit resolution preprocessing was also a necessary step in minimizing a 2CNF formula. If a 2CNF contains 'hidden' unit implicates, one that is not manifested as a unit clause in the formula, using the rest of the algorithm does not minimize the formula either. For instance, consider the 2-Horn-CNF $(\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee b)$, which converts to the following 2CNF implication digraph: $a \rightarrow b$, $b \rightarrow \overline{a}$, $a \rightarrow \overline{b}$, $\overline{b} \rightarrow \overline{a}$. The implication digraph of this formula is already the transitive reduction of itself since there is no graph with fewer arcs that retains the directed paths. But it is obvious that $(\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee b)$ is equivalent to $(\overline{a})$, which is shorter both in the number of clauses and the number of literal occurrences.

Upon closer inspection, we can see that this scenario arises when there is a directed path leading from a vertex p_i to its complement vertex $\overline{p_i}$, but the two are not in a cycle. In order for the formula to be satisfiable, we see that the source vertex p_i must be assigned false, and the destination vertex $\overline{p_i}$ must be assigned *true*. This dictates the truth value in the 2CNF formula of the variable represented by the vertex p_i, which can be expressed as an unit clause and is an unit implicate.

Analysis of the algorithm

The running time of constructing the implication graph of a 2CNF formula is $O(\ell)$, where ℓ is the length of the input. For the next steps in our algorithm, Aho et al. proved that the running time of computing the transitive reduction of a directed graph is at most a constant factor from the running time of Boolean matrix multiplication, which is in turn at most a constant factor from computing the transitive closure of a graph. This means that for a graph with n vertices, if there is an algorithm to compute the transitive closure of an vertex graph in time $O(n^\alpha)$, then there is an algorithm to compute transitive reduction in time $O(n^\alpha)$ (Theorem 3 of [1]). The time complexity result holds true for implication graphs of 2CNF formulas. For the final step, the running time of converting an implication graph back to a 2CNF formula is $O(E)$, where E is the number of arcs. Since $|E| \leq 2m$, $O(E) = O(2m) = O(m)$. Therefore, the total running time of minimizing a 2CNF formula is $O(n^\alpha + m)$, where n is the number of variables, and m is the number of clauses.

10 Minimization of Horn CNF with 2-Pure-Horn and negative clauses

In this section we examine yet another subclass of Horn formulas for which term minimization is in polynomial-time, *Horn CNF with 2-Pure-Horn and negative clauses*. In this subclass, a Horn CNF can have any number of negative clauses of any lengths, as long as each of the pure Horn clauses have a maximum of 2 literals.

According to Theorem 6.3, an irredundant and prime Horn formula can be divided into two parts, the pure Horn component and the negative restriction. Moreover, it was proven in [13] that all negative restrictions of f contains the same number of clauses. It follows then that term minimization of a prime and irredundant Horn CNF really only requires minimization of the pure Horn component.

Since any Horn CNF can be reduced to an equivalent prime and irredundant Horn CNF in quadratic time [13], if the minimization of the pure Horn component can be done in P, then minimization of the original Horn CNF is in P. This is indeed the case when the pure Horn component contains pure Horn clauses of 2 literals each. As shown in the previous section, the minimization of 2-Horn CNF is in P. Therefore the following polynomial-time algorithm can minimize a Horn CNF with 2-Pure-Horn and negative clauses:

1. Reduce the Horn CNF to an equivalent prime and irredundant Horn CNF

2. Reduce the pure Horn component, consisting of only pure Horn clauses of 2 literals

Step one can be done using Hammer and Kogan's algorithm in [13], which takes $O(n^2)$, where n is the number of literal occurrences. Step two can be done using the implication graph reduction technique as shown in the previous section, taking $O(n^\alpha + m)$, where n is the number of variables, and m is the number of clauses.

11 Minimization of Quasi-Acyclic Horn formulas

Quasi-Acyclic Horn formulas [14] are yet another subclass of Horn formulas for which minimization is in polynomial-time. The class of quasi-acyclic Horn formulas actually encompass the subclasses of Horn formulas already presented in previous sections. In order to discuss properties and characteristics of quasi-acyclic Horn formulas, it is necessary to first introduce several concepts:

To a definite Horn CNF formula F, an associated directed implication graph $G(F)$ is constructed as follows: Each vertex in $G(F)$ corresponds to a variable in F, and for each clause $(p \vee \overline{q_1} \vee \overline{q_2} \vee \ldots \vee \overline{q_x})$, arcs $q_1 \rightarrow p, q_2 \rightarrow p, \ldots, q_x \rightarrow p$ are added. Note that this graph construction method is identical to Dowling and Gallier's method for definite Horn clauses in [9].

Building on the concept of implication graphs, the acyclicity of Horn functions can be defined:

Definition 11.1 *(Definition 5.1 of [14])*

- *A definite Horn CNF F is called* acyclic *if and only if the associated graph $G(F)$ does not contain directed cycles.*
- *A definite Horn function f is called* acyclic *if and only if it has an acyclic Horn CNF.*
- *A Horn function f is called acyclic if and only if its definite Horn component $h(f)$ is acyclic.*

Acyclic Horn functions have some important properties. First, checking whether a Horn CNF is acyclic can be done in quadratic time (Corollary 5.4 of [14]). Secondly, different CNF representations of an acyclic Horn function can be reduced to the same irredundant prime CNF. In other words, any acyclic Horn function has an unique irredundant and prime CNF. Moreover, this irredundant and prime CNF is the literal minimal CNF of the function (Corollary 5.6 of [14]). These two properties combined, means that given an arbitrary Horn CNF formula, we are able to check whether it is acyclic, and if it is, we are able to minimize the number of literal occurrences in it, all in quadratic time (A quadratic time algorithm to reduce any Horn CNF to a prime and irredundant CNF is presented in Section 12.1).

Before we can define quasi-acyclic Horn formulas, one more concept, the *2-condensation* of a Horn CNF must be introduced (Section 6 of [14]). First Hammer and Kogan define a clause to be *quadratic* if it has exactly two literals. Then they define $H^2(f)$ to be the conjunction of all quadratic definite Horn prime implicates of a Horn function f, and $G(H^2(f))$ the implication digraph of $H^2(f)$. Using these concepts, the 2-condensation of a Horn CNF can be defined like the following:

Definition 11.2 *(Definition 6.4 of [14]) Let f be a Horn function, and let $F(f)$ be a prime CNF of f; the* 2-condensation *of $F(f)$ is the CNF $F^c(f)$, obtained by replacing all the variables belonging to each strongly connected component $V_r(f)$ in $G(H^2(f))$ by the same variable v_r; the function $c(f)$ represented by $F^c(f)$ is called a* 2-condensation *of f.*

The concept of 2-condensation is very similar to obtaining the compacted acyclic graph as we did in Section 9. The idea is to substitute the variables belonging to the same strongly connected component in the implication graph with the same variable, thus getting rid of cycles in the digraph.

Now we are ready for the definition of quasi-acyclic Horn functions:

Definition 11.3 *(Definition 7.1 of [14]) A Horn function is called* quasi-acyclic *if and only if its 2-condensation is acyclic.*

An interesting observation is that the determination of whether a prime Horn CNF represents a quasi-acyclic Horn function is only concerned with the definite Horn component, since the implication graph construction only deals with definite Horn clauses. Moreover, the implication graph of a quasi-acyclic Horn function may still contain cycles, but these cycles will contain at most 2 vertices so that the 2-condensation of the graph is acyclic.

Proofs are given in [14] that state the recognition of the quasi-acyclicity of a Horn function can be done in quadratic time, and that the minimization of quasi-acyclic Horn formulas can also be done in quadratic time (Lemma 7.2 and Theorem 7.8 of [14]).

It is easy to see that subclasses of Horn formulas that were discussed in Sections 7, 8, 9 and 10 actually all fall under the class of quasi-acyclic Horn formulas. For Horn formulas containing only unit clauses and purely negative Horn formulas, since they have an empty definite Horn component, they are by definition acyclic. 2-Horn formulas fall under the category of quadratic

Horn formulas, and since the class of quasi-acyclic Horn formulas contains the quadratic Horn formulas, the class of 2-Horn formulas is a subclass of quasi-acyclic Horn formulas. Lastly, 2-Horn formulas with negative clauses are also quasi-acyclic because their definite Horn component is quadratic.

12 Approximation Algorithms

As we have shown in Section 6, both term minimization and literal minimization of general Horn CNF formulas are NP-complete. In this section we will look at two polynomial-time approximation algorithms that reduce a given Horn CNF formula to a shorter equivalent Horn CNF.

12.1 Reducing to an irredundant prime Horn CNF

In [12] and [13], Hammer and Kogan present a quadratic time reduction algorithm that transforms a Horn CNF formula into an equivalent Horn CNF that is prime and irredundant. Recall that a CNF is prime if every clause in it is a prime implicate of the underlying function, and a CNF is irredundant if dropping any clause from it yields a CNF that is not equivalent. In the following sections, we will look at Hammer and Kogan's algorithm, as well as their proof on the length of an irredundant and prime Horn CNF. We must first note that the algorithm is devised to reduce Horn CNF formulas without unit implicates. If a Horn CNF with unit implicates was given as input, according to Lemma 9.2, we can simply find all unit implicates by using unit resolution, then use the following algorithm on the portion that does not contain unit implicates.

12.1.1 The algorithm

The approximation algorithm is a two step process. First, each clause in the input Horn CNF is made prime. Secondly, redundant clauses are omitted from the prime Horn CNF. Central to the approximation algorithm is a linear time implication test, which checks whether a given clause is an implicate of a CNF. The implication test algorithm is as follows:

Given a clause $C = (\bigvee_{x \in P} x) \vee (\bigvee_{x \in N} \overline{x})$, and a CNF F.

1. Substitute the partial truth assignment that makes C false, in other words, assign x to true iff $x \in N$, and x to false

iff $x \in P$, into the CNF F.

2. Check whether the Horn CNF F is satisfiable with the above partial truth assignment.

By definition, the clause C is an implicate of F if and only if $F \rightarrow C$. Therefore, C is an implicate of F if and only if step 2 of the implication test finds the partially assigned CNF F to be unsatisfiable. Furthermore, since the satisfiability of a Horn CNF can be determined in linear time $O(n)$, where n is the number of literal occurrences [9], the implication test runs in linear time also.

Using this implication test algorithm, stage 1 of the reduction process reduces each clause to a prime implicate as follows:

```
Given a Horn CNF F
while there exists an unmarked clause C in F
   while there exists a unmarked literal x in C
      remove x from C to produce C';
      if C' is an implicate of F
         C := C';
      else
         mark x in C;
   mark C in F;
```

This stage of the algorithm is straight forward. Recall that a prime implicate is one such that dropping a literal from it produces a clause that is not an implicate. This algorithm simply iterates through every clause in the Horn CNF (every clause in a CNF is an implicate of the underlying function), and attempts to drop each literal in the clause. A literal is only permanently removed from a clause if the clause remains an implicate without it. Hence the every literal in each resulting clause is necessary to make the clause an implicate, making the resulting Horn CNF prime.

The running time for this algorithm is $O(n^2)$, where n is the number of literals. This results from doing the linear time implication test as many times as there are literals in the CNF.

The second stage of the algorithm reduces a Horn CNF into an irredundant equivalent Horn CNF as follows:

```
Given a Horn CNF F
while there exists an unmarked clause C in F
   remove C from F to produce F';
```

```
if C is an implicate of F'
   F := F';
else
   mark C in F;
```

This part of the algorithm removes every redundant clause from F. Since for every clause C in F, $F \rightarrow C$; if $(F \backslash C) \rightarrow C$, then $F \equiv (F \backslash C)$. This algorithm runs in $O(mn)$, where m is the number of clauses in F, and n is the number of literals in F. This results from doing the $O(n)$ implication test m times.

After the 2 stage algorithm, a Horn CNF is transformed into one that is both prime and irredundant. In the next section, we will show Hammer and Kogan's proof that a prime and irredundant Horn CNF is at most a certain length away from the minimal equivalent Horn CNF, both in the number of literal occurrences and the number of clauses.

12.1.2 Length of irredundant and prime Horn CNF

In this section, we present Hammer and Kogan's result on the relationship between the length of an irredundant and prime Horn CNF and the length of an equivalent minimal CNF, both in the number of terms and in the number of literal occurrences.

Theorem 12.1 *(Theorem 5.1 of [13]) If F is an arbitrary irredundant Horn CNF of a Horn function f over n variables, then the number of clauses in F is at most $(n-1)$ times the number of clauses in F', where F' is a term minimal CNF of f.*

The approximation algorithm also reduces the number of literal occurrences in a Horn CNF to a certain bound:

Corollary 12.2 *(Corollary 5.2 of [13]) If F is an arbitrary irredundant Horn CNF of a Horn function f over n variables, then the number of clauses in F is at most $\binom{n}{2}$ times the number of literal occurrences in F', where F' is a literal minimal CNF of f.*

The theorem and the corollary are proven in Section 5 of [13]. The proof builds on the forward-chaining procedure and the number of clauses that must be used in each iteration of the forward-chaining procedure.

12.2 Iterative decomposition

The second approximation algorithm, iterative decomposition [5] is concerned with minimizing the number of clauses in a Horn CNF. The algorithm is largely based on results presented in [12]. In particular, Corollary 4.4 and Corollary 5.4 of [12], restated in [5] as Theorems 3.1 and 3.2, respectively, lay the foundation for the iterative decomposition algorithm:

Theorem 12.3 *(Theorem 3.1 of [5]) Let F_1 and F_2 be two prime CNFs of a Horn function f. Then the conjunction of all definite Horn clauses in F_1 and the conjunction of all definite Horn clauses in F_2 are equivalent (and the function they represent is called the pure Horn component of f).*

Theorem 12.4 *(Theorem 3.2 of [5]) Let F_1 and F_2 be two irredundant and prime CNFs of a Horn function f. Then F_1 and F_2 contain the same number of negative clauses.*

Based on these two theorems, as long as we are working with prime and irredundant Horn CNF formulas (and we can reduce any Horn CNF formula to be prime and irredundant in quadratic time as shown in Section 12.1), in order to minimize the number of clauses, we can simply focus on minimizing the number of clauses in the definite Horn part.

The iterative decomposition algorithm operates on the definite Horn component of a prime and irredundant Horn CNF, and essentially divides the definite Horn component into two smaller conjunctions. Of these two parts, one conjunction cannot be term minimized any further, so the minimization of the number of clauses in the definite Horn component is reduced to minimizing the other conjunction. Therefore the iterative decomposition algorithm isolates a subset of definite Horn clauses so that one needs only to term minimize this subset to term minimize the entire CNF.

At the core of the algorithm is a switching of variables. The basic idea is that the algorithm replaces some of the propositional variables with their complements (replacing v with $\overline{v}$ and $\overline{v}$ with v) so that the resulting CNF remains Horn, and more importantly, contains some negative clauses that can be set aside. A simplified example of this switching is illustrated below. Supposed we have a irredundant and prime definite Horn CNF $F = (\overline{a} \vee \overline{b} \vee c) \wedge (\overline{a} \vee \overline{b} \vee d) \wedge (\overline{a} \vee \overline{d} \vee b)$. By switching on the variable c, we get $F' = (\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee d) \wedge (\overline{a} \vee \overline{d} \vee b)$, which is still Horn, and now contains one negative clause. It is obvious that we can term minimize F', then revert

the switching of variable c in the term minimal CNF so that the resulting formula would be a term minimal CNF that is equivalent to F. Moreover, because F' has fewer definite Horn clauses than F, minimizing the number of terms in F' would be computationally easier than minimizing the number of terms in F.

A naive version of the iterative decomposition algorithm would attempt to switch variables that exist as the positive literal in each clause (these are called heads of the clauses). In almost all but the most trivial definite Horn CNF formulas, this switching causes a domino effect throughout the formula. This occurs when the variable to be switched exists in other clauses as negative literals. In such case, these other clauses would have two positive literals, requiring the switching of the original positive literal to stay Horn. Let us refer back to the simplified example above. Suppose we were to switch the variable d so that the second clause can be made negative. This causes the third clause to become $(\overline{a} \vee d \vee b)$, which demands the switching of b to remain Horn. In this particular case, the subsequent switching on b renders the second clause definite Horn again. (The net effect of this switching basically replaces the variable b with d, and vice versa). Hence this switching of d cycles, and for our purposes, d is considered unswitchable.

In [5], the authors provide a brute-force switching algorithm that attempts to switch the head of a clause until it either succeeds or finds that the variable is unswitchable. Then the above algorithm is used as a subroutine in another algorithm that does this for every clause in the CNF. This algorithm returns the set of clauses that can be made negative by switching of some variables, and the remaining unswitchable definite Horn CNF. This naive algorithm takes $O(n^2m)$, where n is the number of literal occurrences and m is the number of clauses.

The authors then continue to show that the order in which the variables are switched did not matter to the end result, and that switchable clauses can be identified and the order of switching can be chosen more intelligently, so that the number of iterations is minimized. This improved decomposition algorithm involved constructing a directed graph from the definite Horn CNF using the exact method as in [14], identifying the strongly connected components in the graph, and then identifying which clauses are switchable using information gathered from the strongly connected components. (Please refer to [5] for details) The new algorithm, called GRAPH-ALGO in [5] runs in $O(n)$, where n is the number of literal occurrences.

The iterative decomposition algorithm by itself does not minimize the number of clauses a given Horn CNF formula. Rather, it identifies a smaller portion of a given Horn CNF formula as the only part that needs to be minimized in order to minimize the entire formula. Therefore, it can be used as a preprocessor to other minimization algorithms, reducing the size of the formula that needs to be minimized.

For example, given a Horn formula F, we first process it to get an equivalent CNF that is prime and irredundant, $F' = F_H \wedge F_N$, where F_H is definite Horn and F_N is negative. We know that to minimize the number of clauses in F', we only need to minimize the number of clauses in F_H because every prime and irredundant CNF of the same Horn function has the same number of negative clauses. Then we can use the iterative decomposition algorithm to split F_H into two parts, $F_H = F_{H1} \wedge F_{H2}$, where term minimizing F_H amounts to term minimizing F_{H1} (there is a possibility that $F_H = F_{H1}$ and F_{H2} is empty). So in the end, to minimize the number of clauses in F, we only need to minimize the number of clauses in F_{H1}, which is an computationally easier task if F_{H1} is shorter than F_H.

References

[1] Aho, A. V., Garey, M. R. and Ullman, J. D., *The transitive reduction of a directed graph*, SIAM Journal of Computing, pp. 131-137, June 1972.

[2] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The design and analysis of computer algorithms*, Addison-Wesley, MA, 1974.

[3] Aspvall, B., Plass, M. F. and Tarjan, R. E., *A Linear-time algorithm for testing the truth of certain quantified Boolean formulas*, Information Processing Letters, pp. 121-123, March 1979.

[4] Boros, E. and Čepek, O., *On the complexity of Horn minimization*, Rutcor Research Report, 1-94, January 1994.

[5] Boros, E., Čepek, O., and Kogan, A., *Horn Minimization by Iterative Decomposition*, DIMACS Technical Report 97-34, July 1997.

[6] Cook, S. A., *The Complexity of theorem-proving procedures*, Proceedings of the 3rd ACM Symposium on Theory of Computing, pp. 151-158, 1971.

[7] Dalal, M. and Etherington, D. W., *A hierarchy of tractable satisfiability problems*, Information Processing Letters 44, pp. 173-180, 1992.

[8] Davis, M., Logemann, G. and Loveland, D., *A machine program for theorem proving*, Communications of the ACM 5: pp. 394-397, 1962.

[9] Downling, W. F. and Gallier, J. H., *Linear-time algorithms for testing the satisfiability of propositional Horn formulae*, J. Logic Programming, pp. 267-284, 1984.

[10] Even, S., Itai, A. and Shamir A., *On the complexity of timetable and multi-commondity flow problems*, SIAM Journal of Computing 5(4), pp. 691-703, 1976.

[11] Garey, M. and Johnson, D., *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[12] Hammer, P. L. and Kogan, A., *Horn functions and their DNFs*, Information Processing Letters, pp. 23-29, November 1992.

[13] Hammer, P. L. and Kogan, A., *Optimal compression of propositional Horn knowledge bases: complexity and approximation*, Artificial Intelligence 64, pp. 131-145, 1993.

[14] Hammer, P. L. and Kogan, A., *Quasi-acyclic propositional Horn knowledge bases: Optimal compression.*, IEEE Transactions on Knowledge and Data Engineering, 7(5):751-762, 1995.

[15] Hemaspaandra, E. and Wechsung, G., *The minimization problem for Boolean formulas*, SIAM Journal of Computing, vol. 31, no. 6, pp. 1948-1958, 2002.

[16] Jones, N. D. and Laaser, W. T., *Complete problems for deterministic polynomial time*, Theor. Comp. Sci. 3:107-117, 1977.

[17] Meyer, A. and Stockmeyer, L., *The equivalence problem for regular expressions with squaring requires exponential space*, Proceedings of the 13th IEEE Symposium on Switching and Automata Theory, pp. 125-129, 1972.

[18] Reingold, E. M., Nievergelt, J. and Deo, N., *Combinatorial algorithms: theory and practice*, Prentice-Hall, NJ, 1977.

[19] Robinson, J. A., *A machine-oriented logic based on the resolution principle*, Journal of the ACM, 12: pp. 23-41, 1965.

[20] Schaefer, T. J., *The complexity of Satisfiability problems*, Proceedings of the 10th ACM Symposium on Theory of Computing, pp. 216-226, 1978.

[21] Scutellà, M. G., *A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability*, Journal of Logic Programming 8: pp. 265-273, 1990.

[22] Tarjan, R., *Depth first search and linear graph algorithms*, SIAM Journal of Computing, 1(2):146-160, June 1972.

[23] Umans, C, *The minimum equivalent DNF problem and shortest implicants*, Proceedings of the 39th IEEE Symposium on Foundations of Computer Science, IEEE Computer Science Press, Los Alamitos, CA, pp. 556-563, 1998.

[24] Umans, C, *Hardness of approximating* Σ_2^p *minimization problems*, Proceedings of the 40th IEEE Symposium on Foundations of Computer Science, IEEE Computer Science Press, Los Alamitos, CA, pp. 465-474, 1999.