Solving SATISFIABILITY with Molecular Algorithms

by

David Carley

Master of Science Project

Presented to the Faculty of the Graduate School of

Rochester Institute of Technology

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Chair:
Dr. Christopher Homan

Reader:
Dr. Stanisław Radziszowski

Observer:
Dr. Reynold Bailey

Draft: June 28, 2012

Abstract

Molecular computation uses techniques from molecular biology and combinatorial chemistry to perform computation. We explore, via simulation, three distinct molecular algorithms for solving SATISFIABILITY. The simulation measures the number of molecular operations each algorithm needs to solve SATISFIABILITY. The test input consists of a set of random 3-SAT instances distributed over a range of clause-variable ratios ($\alpha = [0.2, 14.0]$).

Contents

Chapter 1

Introduction

Molecular computation uses interactions between genetic molecules, such as DNA or RNA, to perform computational tasks. We provide an experimental system for simulating three molecular algorithms. In this chapter we discuss the advantages of molecular computation versus standard computation. This discussion includes an introduction to the simulation of molecular algorithms. We conclude the chapter with an overview of the contents of this report.

1.1 Introduction to molecular computation

NP problems, such as SATISFIABILITY, may be verified in polynomial time with the aid of a short (i.e. of length polynomial in the length of the input) proof called a *witness*; NP problems may be solved by checking all possible *witness candidates*. In a standard computational environment, brute force search checks all witness candidates in exponential time.

Molecular computation requires exponential space in order to represent all witness candidates. This combinatorial space of witness candidates can be filtered in polynomial

time by parallel molecular operators.

A Boolean formula ϕ is said to be in Conjunctive Normal Form (CNF) if ϕ consists of a conjunctive set of Boolean disjunctive clauses. (Throughout this report we use CNF instances ϕ with n variables, m clauses, and k literals.) We have, e.g.,

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each clause C_i contains k disjunctive Boolean literals

$$C_i = (v_1 \vee v_2 \vee \cdots \vee v_k).$$

A witness for a SATISFIABILITY instance is a Boolean assignment to the variables that makes the formula true. Such a witness can be represented as a vector B in $\{0,1\}^n$, where n is the number of variables in ϕ, as follows. Let $\{v_1, \ldots, v_n\}$ be the variables of ϕ, for each i in $\{1, \ldots, n\}$, the ith element of B is denoted B_i and represents an assignment of true or false to v_i. A witness candidate for a SATISFIABILITY instance may be verified in polynomial time with CHECKSAT(ϕ, B) (Algorithm 1.1.1 below).

Algorithm 1.1.1: CHECKSAT(ϕ, B)

```
for each clause C in φ
  do { test_clause ← False
       for each literal ℓ in C
         do { if B satisfies ℓ
              then test_clause ← True
       if test_clause = False
         then return (False)
return (True)
```

Figure 1.1: CHECKSAT(ϕ, B) iterates over each of the clauses C in the CNF instance ϕ. The bit-vector B encodes a witness candidate for the CNF instance ϕ. The *test_clause* variable gets set to $False$, assuming that the clause cannot be satisfied. If the clause can be satisfied with the input configuration B, then the algorithm continues. If each of the m clauses can be satisfied, then CHECKSAT(ϕ, B) returns $True$; otherwise the algorithm returns $False$.

CHECKSAT(ϕ, B) may be used as a subroutine in a brute force SATISFIABILITY solver. Algorithm 1.1.2 provides pseudocode for a brute force SATISFIABILITY solver BRUTESAT(ϕ).

In this project, we consider molecular algorithms to solve SATISFIABILITY. Molecular algorithms permit parallelism on a massive scale [1, 16]. Molecular operations, such as *append* or *extract*, can perform in parallel on all of the string contents of a test tube [1, 16, 13]. In Chapter 3, we explore techniques from combinatorial chemistry to generate combinatorial sets [16, 10, 13].

Algorithm 1.1.2: BRUTESAT(ϕ)

```
// Input φ consists of n variables.
// Bit-vector B represents a witness candidate.

for each B ∈ {0,1}^n
  do { if CHECKSAT(φ, B)
       then return (SATISFIABLE)
return (UNSATISFIABLE)
```

Figure 1.2: BRUTESAT(ϕ) tests a maximum of 2^n Boolean witness candiates, using the CHECKSAT(ϕ, B) algorithm. If the witness candidate B satisfies the input instance ϕ, then the algorithm returns `SATISFIABLE`; otherwise the algorithm returns `UNSATISFIABLE`.

Algorithm 1.1.3: EXTRACTSAT(ϕ)

```
// Input φ consists of n variables.

T ← COMBINATORIALGENERATE(n)
for each clause C in φ
  do { T_C ← ∅
       for each literal ℓ in C
         do { T_T ← extract(T, ℓ)
              T_C ← mix(T_C, T_T)
       T ← T_C
if T = ∅
  then return (UNSATISFIABLE)
return (SATISFIABLE)
```

Figure 1.3: EXTRACTSAT(ϕ) collects satisfying Boolean literals from each clause in ϕ. Initially, EXTRACTSAT(ϕ) constructs a combinatorial space T using the subroutine COMBINATORIALGENERATE(n) which we introduce in Chapter 3. The initial space T contains string configurations representing all potential witness candidates for ϕ. The space T gets filtered down to witness all clauses. These potential solutions are incrementally mixed into the tube T_C for each clause.

Let us consider Algorithm 1.1.3 as a simplified version of Lipton's algorithm [16, 13]. The ExtractSat(ϕ) function provides an introductory view of a molecular algorithm. ExtractSat differs in how the algorithm validates each candidate. The brute force algorithm, BruteSat, generates sequentially an exponential number of witness candidates. On the other hand, exponential witness canidates with ExtractSat get filtered in parallel.

1.2 Simulation of molecular Satisfiability solvers

We consider three molecular algorithms for solving Satisfiability: Lipton's [16], Ogihara and Ray's [19, 20], and a new algorithm, introduced here, that we call the 'Distribution' algorithm. Lipton's algorithm begins with a combinatorial space of all n-bit witness candidates and filters the combinatorial space so that only those that satisfy the input formula remain. Ogihara and Ray's algorithm constructs a space of witness candidates using heuristic search. The Distribution algorithm expands a set of witnesses with non-conflicting literals from each clause. Chapters 3 and 4 discuss the implementation of these algorithms.

This project introduces a Satisfiability solver framework for molecular algorithms, which we call 'Molecular Simulation'. This system provides standard operations for molecular computation which we introduce in Chapter 2. It also records runtime metrics, including counts of molecular operators, memory footprints, and execution times. These metrics let us analyze the algorithmic performance of each molecular algorithm.

Molecular Simulation automates execution of DIMACS CNF instances. It measures key properties for a set of randomly generated 3-Sat instances. The 3-Sat instances span discrete clause-variable ratios from 0.2 to 14.0 in increments of 0.2, creating a sweep of Satisfiability instances. This experimental setup generates Satisfiability problem instances with both `SATISFIABLE` and `UNSATISFIABLE` configurations.

1.3 Report Overview

In the following chapters, we describe molecular algorithms for solving Satisfiability. We begin Chapter 2 with an introduction to gene sequencing technologies and molecular biology. We define molecular operations for operating on DNA or RNA. Next, we introduce Satisfiability as a language and as a Boolean circuit.

Chapters 3 and 4 introduce each of the three molecular algorithms for solving Satisfiability. In Chapter 3, we discuss Lipton's [16, 13] and Ogihara and Ray's [19, 20, 26] algorithms for Satisfiability. The chapter concludes with a discussion of existing simulation frameworks and physical implementations of these molecular algorithms. Chapter 4 introduces the Distribution algorithm.

Chapters 5 and 6 discuss the project implementation. In Chapter 5, we introduce our software, Molecular Simulation, for simulating molecular algorithms. Chapter 6 describes the experimental workflow for importing Satisfiability instances for each of the three molecular algorithms we study.

Chapter 7 provides a discussion of algorithm performance based test results. Chapter 8 concludes with a summary of contributions of this project and future directions for molecular computation.

Chapter 2

Background

This chapter provides a background on molecular computation techniques. We begin with an introduction to nanotechnology and then provide an example of how information is encoded using molecular matter. Following this example, we introduce Adleman's molecular operators for solving an instance of HAMILTONIAN PATH. The operators provide an instruction set for molecular computation, and provide the primitives for constructing molecular algorithms.

In the second half of this chapter, we provide an introduction to SATISFIABILITY. We define SATISFIABILITY as a circuit. We then view SATISFIABILITY as a language. We also discuss practical matters related to efficiently evaluating SATISFIABILITY, such as how to encode input and output, and how to classify instances of SATISFIABILITY in the tests that we perform.

2.1 On nanotechnology and construction of molecules

Richard Feynman founded the field of nanotechnology in his 1959 talk "There's Plenty of Room at the Bottom" [7]. Examples of applied nanotechnology include the manufac-

turing of graphene [24] and DNA nanopores [17]. Graphene consists of a planer arrangement of carbon atoms that provides desirable physical and electrical properties [24]. DNA nanopores use graphene to create a physical channel for reading genetic sequences [11]. Gene sequencing technologies provide an example of applied nanotechnology [11, 15, 21].

Smaller and cost-effective DNA sequencers provide the ability to read the contents of a gene. Benchtop sequencers [15, 21] allow doctors to treat patients at the genome level from their office. Life Technologies and Oxford Nanopore offer gene sequencers based on solid-state semiconductor technology [15, 21].

2.2 Genetic encoding schemes

Microbiology studies the interactions among organic molecules. In this project, we explore the use of applied genetics as a means for generalized computation. Molecular computation encodes data as sequences of DNA or RNA.

Arbitrary encodings that represent mappings from variables to physical oligonucleotides may have undesirable structure and functionality. Conventional techniques from molecular computation employ variable mappings from a library of oligonucleotides.

An *oligonucleotide* is a short string of genetic information. There are several configurations for DNA and RNA; these include +RNA, −RNA, +DNA, −DNA, ±RNA, ±DNA, and +mRNA [2]. The polarity of DNA denotes the direction of the genetic information. '+DNA' is denoted $5'$—$3'$ and '−DNA' is denoted $3'$—$5'$. We focus on +DNA and −DNA as the substrate for computational states. The computational states, in our setting, encode candidate witnesses for SATISFIABILITY.

Suppose for example that we would like to encode the sequence of integers $S = [1, 3, 4, 3, 2, 0]$ as an equivalent oligonucleotide representation with the definitions in Table 2.1. Gene sequencing tools permit one to read and decode data according to Table 2.1.

Table 2.1: A mapping of the integers $[0, 4]$ with arbitrary oligonucleotide definitions.

Integer	**Oligonucleotide**	**Reverse-complement**
0	$5'$TCTCCC$3'$	$3'$AGAGGG$5'$
1	$5'$AAACCC$3'$	$3'$TTTGGG$5'$
2	$5'$GGTAAA$3'$	$3'$CCATTT$5'$
3	$5'$CCCTCC$3'$	$3'$GGGAGG$5'$
4	$5'$CTTTTC$3'$	$3'$GAAAAG$5'$

The resulting oligonucleotide O is defined as

$$O = 5'\texttt{AAACCC} \mid \texttt{CCCTCC} \mid \texttt{CTTTTC} \mid \texttt{CCCTCC} \mid \texttt{GGTAAA} \mid \texttt{TCTCCC}3'.$$

Molecular computation uses oligonucleotides for both storing and operating on a problem state. These operations include matching and replication. Although this report describes artificial processes, DNA in natural settings undergoes the same transformations that we exploit here. Interactions between genetic molecules are the fundamental mechanism for generic computation with oligonucleotides.

In the following chapters, we describe molecular algorithms for SATISFIABILITY. In the next section, we introduce techniques from Adleman's molecular toolbox [1].

2.3 Adleman's molecular toolbox for solving HAMITONIAN PATH

In 1994, Leonard Adleman performed the first molecular computation using recombinant DNA in a bench laboratory setting [1]. This experiment solved a six vertex instance of HAMILTONIAN PATH, an NP-complete problem. In this section, we describe the techniques used in this experiment. We provide definitions for the following operations from Adleman's

molecular toolbox: append, extract, mix, split, and purify.

Definition 2.3.1. HAMILTONIAN PATH

Given an undirected graph G, does there exist a path that visits every vertex exactly once?

Adleman uses oligonucleotides for defining each vertex for encoding a graph. His scheme for encoding a graph's vertices shares a similar definition from our example of encoding a sequence of integers, given in Table 2.1. Representing edges requires a reverse-complement oligonucleotide, which connects the suffix of the vertex v_i with the prefix of v_j. Let us consider an example. Let

$$v_1 = 5'\texttt{ATCTTT}3'$$

$$v_2 = 5'\texttt{CCTATA}3'.$$

From the definition of v_1 and v_2, we can construct an edge $e_{1,2}$ as

$$e_{1,2} = 3'\texttt{AAAGGA}5'.$$

Appending v_2 to v_1 is accomplished by first attaching the edge $e_{1,2}$ to the vertex v_1

$$5'\texttt{ATCTTT}3'$$

$$3'\texttt{AAAGGA}5'.$$

Next we attach v_2 to the resulting complex, yielding

$$5'\texttt{ATCTTT}|\texttt{CCTATA}3'$$

$$3'\texttt{AAAGGA}5'.$$

Finally the edge may be removed and we have the sequence

$$v_1 \cdot v_2 = 5'\texttt{ATCTTT}|\texttt{CCTATA}3'.$$

The sequence $v_1 \cdot v_2$ represents the path v_1 to v_2, and can be obtained with the *append* operation. A test tube T stores witness candidates. The tube T starts as an empty tube. To solve HAMITONIAN PATH, we introduce equimolar portions of each oligonucleotide vertex for a starting configuration, using the *mix* operation.

Definition 2.3.2. Mix *combines n test tubes of information.*

$$T \leftarrow \text{mix}(T_1, \ldots, T_n)$$

The output consists of a single set $T = T_1 \cup \cdots \cup T_n$.

A small initial set may be amplified using *polymerase chain reaction* (PCR). PCR thermocycles the contents of the tube to replicate the contents. Introducing each vertex representation to the contents randomly generates all potential paths. A set of DNA configurations are generated to represent the set of all witness candidates for Hamiltonian Paths in a graph instance. This set of DNA configurations will be filtered to only include configurations that witness Hamiltonian Paths in G.

Append attaches a string to each string contained in a test tube. *Split* portions a tube into multiple portions. In Chapter 3, we will use split-mix synthesis as a means for generating a combinatorial space.

Definition 2.3.3. Append *concatenates each element in* T *with the oligonucleotide* s.

$$T' \leftarrow \text{append}(T, s)$$

Definition 2.3.4. Split *portions T into two tubes.*

$$[T', T''] \leftarrow \text{split}(T)$$

Each of the resulting tubes, T' and T'', contain the same representative elements of T.

The initial and terminal conditions for the graph get fulfilled by extracting, from the tube T, only paths that begin with V_{in} and end with V_{out}. Extracting only strings from T that match these conditions constrain the number of potential strings to only those that satisfy the conditions of the graph instance.

Definition 2.3.5. Extract *separates all oligonucleotides from T containing the sequence s.*

$$T' \leftarrow \text{extract}(T, s)$$

The output consists of a set T' of those oligonucleotides containing s.

The tube T consists of possible encodings that have the correct starting and ending vertices. We select only strings of length n, where n is the number of vertices in G, to ensure that all vertices get traversed. This can be performed using *gel electrophoresis*, a technique for sorting molecules by mass.

Next, we ensure that each vertex occurs exactly once. Introducing reverse-complement oligonucleotides for each vertex to the set of witness candidates binds to the respective vertex. If a vertex occurs multiple times in a path, then the string representation gets discarded. This process ensures each vertex corresponds to a potential Hamiltonian Path.

Once all of the vertices have been filtered, we check T using *detect* to determine if any valid paths remain. If valid paths exist, then the oligonucleotide from T may be read for the path assignment.

Definition 2.3.6. Detect *determines if any encodings are present in* T.

$$\text{detect}(T)$$

The output consists of 'true' or 'false' for $T \neq \emptyset$ *or* $T = \emptyset$*, respectively.*

2.3.1 Additional molecular operators

In the following chapters, we will use the molecular operators for constructing molecular SATISFIABILITY solvers. The Distribution algorithm, introduced in Chapter 4, requires the *splice* operation.

Definition 2.3.7. Splice *cuts an oligonucleotide* $a = a_1 \cdot b \cdot a_2$ *with a subsequence* b *into two pieces by a restriction enzyme.*

$$[a_1, a_2] \leftarrow \text{splice}(a, b)$$

These two pieces are a_1 *and* a_2.

In the implementation of a simulation system, we avoid redundant string representations with the *purify* operation. This is a synthetic version of PCR. Purify balances the space representation of molecules with a uniform distribution.

Definition 2.3.8. Purify *provides a uniform distribution from the contents of* T *as* T'.

$$T' \leftarrow \text{purify}(T)$$

2.4 Definition of SATISFIABILITY

Definition 2.4.1. SATISFIABILITY

$$\text{SATISFIABILITY} = \{\langle\phi\rangle \mid \phi \textit{ is a satisfiable Boolean formula}\}[23].$$

Cook and Levin introduced independently the canonical instance of an NP-complete language SATISFIABILITY [4, 14]. An NP-complete language is one that is in NP and NP-hard. An NP-hard language is a decision problem that can be reduced in polynomial time from any NP language [23]. NP-hard problems includes the HALTING PROBLEM [23], which asks if a program on a given input can terminate. Witnesses for SATISFIABILITY or any other NP problem are of length polynomial in the length of the input ϕ.

Standard forms for SATISFIABILITY include Boolean CNF, k-CNF, and k-SAT problem definitions.

Definition 2.4.2. CNF *is a Boolean formula that consists of the conjunction of sets of disjunctive literals.*

Definition 2.4.3. k-CNF *is a CNF Boolean formula where each disjunctive clause contains k literals.*

Definition 2.4.4. k-SAT *is a problem variant of* SATISFIABILITY *where the satisfiable instances are exactly k-CNF satisfiable.*

One way to validate a SATISFIABILITY instance is to input witness candidates to a circuit. Let us consider a circuit for a SATISFIABILITY instance having three levels. This circuit consists of n inverters, m **OR** gates, and one **AND** gate with m-fan-in. This circuit behaves according to the internal wiring of the input CNF instance ϕ. Figure 2.1 contains a schematic for SATISFIABILITY.

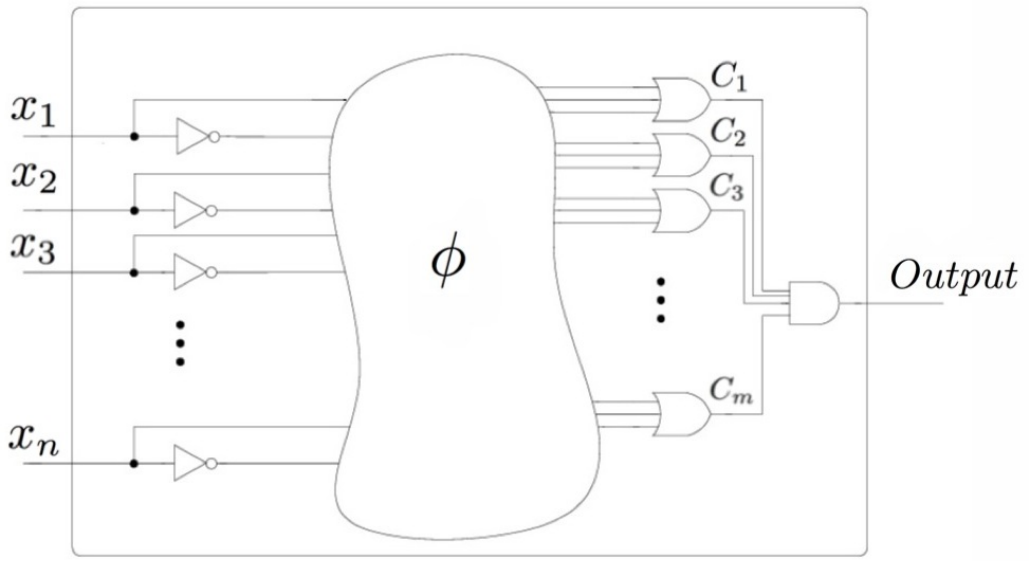

Figure 2.1: A circuit describing SATISFIABILITY.

The realization of SATISFIABILITY as a circuit reveals two aspects of this problem. SATISFIABILITY can be implemented as a circuit with the number of gates proportional to the problem size. The worst case verification for all 2^n possible witness candidates may be performed with the circuit in Figure 2.1. This circuit consists of the hardware equivalent version of the SATISFIABILITY validator CHECKSAT, as shown in Chapter 1.

2.5 Evaluating SAT Solvers

In this section, we describe the standards for encoding the SATISFIABILITY problem that we adopt. The standards come from from the SATISFIABILITY Competition [5, 22].

Next, we introduce a problem instance classification scheme for SATISFIABILITY. Classification of SATISFIABILITY problem instances include random, combinatorial, and industrial SAT instances [22]. Our experiment in Chapter 6 generates random k-SAT inputs.

2.5.1 Input and output

The SAT Competition ranks implementations of solvers for evaluating SATISFIABILITY [22]. SAT solvers are evaluated on three categories of input: industrial, combinatorial, and random instances. The input and output standards for SATISFIABILITY allow common benchmarks for SAT solvers. We conform to the standards of this competition `http://www.satcompetition.org/`.

Input

DIMACS CNF provides a standard input for SATISFIABILITY [5]. The format permits sharing of existing SATISFIABILITY benchmarks by encoding SATISFIABILITY in conjunctive normal form (CNF). We provide an example of this encoding in Section 5.6.

Output

SAT Competition output consists of the status of a DIMACS CNF input instance [22]. The status is provided for the instance as either `SATISFIABLE`, `UNSATISFIABLE`, or `UNKNOWN`. If an instance can be determined as satisfiable, then a witness satisfying the instance gets included with the output status. We provide an example along with custom output logging in Section 5.7.

2.5.2 Metrics for classifying SATISFIABILITY

SAT phase transition and SAT backbones are two metrics for classifying SATISFIABILITY. We will use these metrics in the next section for defining a collection of random k-SAT instances.

The ratio of m clauses to n variables $\alpha = m/n$ provides a characterization where phase transitions may occur in the space of all k-CNF formula [6, 12]. The SAT phase tran-

sition is a region where both satisfiable and unsatisfiable instances are likely [12]. This region frequently separates trivially satisfiable and frequently over-constrained unsatisfiable SATISFIABLE problem instances. SATISFIABILITY instances with low α are frequently under-constrained and trivially satisfiable; those instances with high α are frequently over-constrained and trivially unsatisfiable [12].

Definition 2.5.1. SAT *backbones are the variable assignments present in all of the satisfying assignments to a* SATISFIABILITY *problem instance [27].*

SAT backbones contain a set of variables that occur in all satisfiable witnesses for an input instance. If there are no such variables in the set of all witnesses for a problem instance, then the set is empty.

2.5.3 SATISFIABILITY instances

There are several methods for generating SATISFIABILITY instances. We consider three classes [22]: random assignment, combinatorial problems, and industrial applications.

Random SAT

A random k-SAT instance [25], for fixed m and n, is one drawn uniformly from the set of all k-CNF formulas having m clauses and n variables.

Hard combinatorial SAT

Combinatorial problem instances are well known difficult benchmark cases. These instances include games and graph theoretic problems represented as SATISFIABILITY.

Industrial SAT

Industrial processes apply SATISFIABILITY to solve real world problems, including circuit layout, planning, logistics, circuit fault testing, and many other industrial NP-complete problems. Industrial SAT applications often apply heuristics and approximation techniques to relax the problem. This allows approximate solutions to be computed efficiently with respect to time.

Chapter 3

Existing molecular algorithms for SATISFIABILITY

In this chapter, we discuss two molecular algorithms for SATISFIABILITY. These algorithms construct sets of all witnesses for SATISFIABILITY instances. Lipton's algorithm requires a combinatorial space of all witness candidates to be constructed and then filters out invalid ones. Ogihara and Ray's algorithm constructs a set of witness candidates throughout execution. Following the description, we explore the physical implementations of—and simulation frameworks for—these algorithms.

Table 3.1: Components of Boolean literals and equivalent literal representations.

Literal	**Variable** (v)	**Polarity** (P)	**DIMACS** ($\pm v$)	**Condensed** (v_P)
x_1	1	T	1	$1_{\mathtt{T}}$
$\neg x_1$	1	F	-1	$1_{\mathtt{F}}$
x_n	n	T	n	$n_{\mathtt{T}}$
$\neg x_n$	n	F	$-n$	$n_{\mathtt{F}}$

The algorithm definitions and example traces use the literal conventions listed in Table

3.1. Table 3.1 lists components of a literal (variable and polarity), along with equivalent forms (DIMACS and a condensed representation).

In Chapter 1, witness candidates for SATISFIABILITY were represented as a bit-vector B. Consider the equivalent representation for witness candidates in Figure 3.1.

$$
\begin{aligned}
B &= [0, 1, 0, 1\rangle \\
L &= \{\neg x_1, x_2, \neg x_3, x_4\} \\
V &= \{1_{\mathtt{F}}, 2_{\mathtt{T}}, 3_{\mathtt{F}}, 4_{\mathtt{T}}\} \\
D &= \mathtt{SFTFT}
\end{aligned}
$$

Figure 3.1: The bit-vector $B = [0, 1, 0, 1\rangle$ can be represented as the set of literal assignments $L = \{\neg x_1, x_2, \neg x_3, x_4\}$. Using the condensed representation from Table 3.1 we can represent the set in condensed form $V = \{1_{\mathtt{F}}, 2_{\mathtt{T}}, 3_{\mathtt{F}}, 4_{\mathtt{T}}\}$. A directed polarity string with initial sequence (`S`), followed by a sequence of literals provides $D = \mathtt{SFTFT}$.

We use the directed polarity string (representation D in Figure 3.1) as shorthand for a directed oligonucleotide. The directed string representation D can be indexed by the variable v. In Lipton's algorithm, literal configurations for a variable v get extracted directly ($v_{\mathtt{T}}$ or $v_{\mathtt{F}}$).

In Ogihara and Ray's algorithm, we extract satisfying literal configurations from an ordered clause (a, b, c). We use the literal configuration v_P to extract the satisfying literal P (either `T` or `F`) for the variable v. We use the literal configuration v_N to extract any other non-satisfiable literal assignments to v.

3.1 Lipton's algorithm for SATISFIABILITY

Introduced in 1995 by Richard Lipton [16], this algorithm filters satisfiable witnesses from a combinatorial space of all witness candidates. Lipton's algorithm is analogous to a

conventional brute-force search for all witnesses of a SATISFIABILITY instance.

LIPTON'S ALGORITHM (Algorithm 3.1.1) first constructs a combinatorial space T containing oligonucleotide configurations for all potential witness candidates. The algorithm iterates over each of the clauses C in ϕ.

From each clause C, each of the literals contained within each clause C get filtered to satisfiable witnesses. The contents of T get filtered by incrementally extracting those literals that satisfy the current clause. The next iteration filters witnesses that satisfy the previous clauses from T and the literal contents from C. The algorithm terminates with a set of witnesses T for the CNF input ϕ. If ϕ is unsatisfiable, then $T = \emptyset$.

Algorithm 3.1.1: LIPTON'S ALGORITHM(ϕ)

$T \leftarrow$ COMBINATORIAL GENERATE(n)
for each clause C in ϕ
do {
  $T_c \leftarrow \emptyset$
  for each literal v in C
  do {
    if v is a positive literal
    then { $T_P \leftarrow \text{extract}(T, v_{\mathtt{T}})$; $T_c \leftarrow \text{mix}(T_P, T_c)$ }
    else { $T_N \leftarrow \text{extract}(T, v_{\mathtt{F}})$; $T_c \leftarrow \text{mix}(T_N, T_c)$ }
  }
  $T \leftarrow \text{purify}(T_c)$
}
return (detect(T))

Figure 3.2: LIPTON'S ALGORITHM iterates over each of the m clauses. The contents of T_C grows incrementally with configurations from T that satisfy the literal v. Once the entire clause C has been evaluated, T_C contains configurations that witness the observed conditions. The contents of T_C are stored as T for the next clause; T now contains configurations that witness all previous clauses. Once complete, the tube T contains all witnesses for ϕ, if any witnesses exist.

3.1.1 Description of Lipton's algorithm

The function COMBINATORIAL GENERATE (Algorithm 3.1.2) implements the split-mix synthesis technique [9, 10]. COMBINATORIAL GENERATE returns a tube T_{comb} consisting of oligonucleotides that represent all 2^n distinct witness candidates. The tube T_{comb} begins with an initial medium denoted by $\mathtt{S}$. An iterative loop extends T_{comb} using split-mix synthesis. Each split corresponds with appending the tubes with a true ($\mathtt{T}$) and false ($\mathtt{F}$) assignment. The two tubes are mixed and purified to contain equimolar portions of each witness candidate.

Algorithm 3.1.2: COMBINATORIAL GENERATE(n)

$T_{comb} \leftarrow \emptyset$
$T_{comb} \leftarrow \text{mix}(T_{comb}, \mathtt{S})$
for $v \leftarrow 1$ to n
do $\begin{cases} [T_1, T_2] \leftarrow \text{split}(T_{comb}) \\ T_1 \leftarrow \text{append}(T_1, v_{\mathtt{T}}) \\ T_2 \leftarrow \text{append}(T_2, v_{\mathtt{F}}) \\ T_{comb} \leftarrow \text{mix}(T_1, T_2) \end{cases}$
$T_{comb} \leftarrow \text{purify}(T_{comb})$
return (T_{comb})

Figure 3.3: COMBINATORIAL GENERATE constructs a combinatorial space consisting of 2^n molecular configurations in polynomial time.

Let us consider an example execution of COMBINATORIAL GENERATE with $n = 2$. The tube T_{comb} begins as an empty tube. A start configuration $\mathtt{S}$ initiates the tube T_{comb} with a medium for combinatorial synthesis.
We begin with the initial contents

$$T_{comb} = \{\mathtt{S}\}.$$

Iteration $v = 1$:

First, split the contents of T_{comb}. We have

$$T_1 = \{\texttt{S}\}, \text{ and}$$
$$T_2 = \{\texttt{S}\}.$$

Next, append each of the tubes with a positive (T) and negative (F) assignment for the literal v_1. We have

$$T_1 = \{\texttt{ST}\}, \text{ and}$$
$$T_2 = \{\texttt{SF}\}.$$

Mix the contents of T_1 and T_2 to form T_{comb} for the next iteration. We have

$$T_{comb} = \{\texttt{ST}, \texttt{SF}\}.$$

Iteration $v = 2$:

Split the contents of T_{comb}. We have

$$T_1 = \{\texttt{ST}, \texttt{SF}\}, \text{ and}$$
$$T_2 = \{\texttt{ST}, \texttt{SF}\}.$$

Next, append each of the tubes with a positive (T) and negative (F) assignment for the

literal v_2. We have

$$T_1 = \{\texttt{STT}, \texttt{SFT}\}, \text{ and}$$
$$T_2 = \{\texttt{STF}, \texttt{SFF}\}.$$

Mix the contents of T_1 and T_2 to form T_{comb} for the final iteration. The algorithm Combinatorial Generate returns the following tube

$$T_{comb} = \{\texttt{STT}, \texttt{SFT}, \texttt{STF}, \texttt{SFF}\}.$$

Combinatorial Generate generates all witness candidates for a Satisfiability instance. Lipton's Algorithm filters, from a combinatorial space T, configurations that represent witnesses for the input ϕ.

3.1.2 Detailed trace of Lipton's algorithm

Appendix B lists a detailed execution trace for Lipton's algorithm.

3.2 Ogihara and Ray's algorithm for Satisfiability

Ogihara and Ray's algorithm (Algorithm 3.2.1) consist of a breadth-first evaluation of clauses from a CNF formula [19, 20]. The algorithm constructs a set of witness candidates based on a parse of a 3-CNF formula. In this section, we describe the preconditions and execution of Ogihara and Ray's algorithm.

Algorithm 3.2.1: OGIHARA AND RAY'S ALGORITHM(ϕ)

```
// Input φ consists of n variables.
// Each clause C contains ordered literals (a, b, c).

T ← {STT, STF, SFT, SFF}
for v ← 3 to n
do { [T_P, T_N] ← split(T)
     for each clause C in φ
     do { (a, b, c) ← C
          if v_T = c
          then { T_P1 ← extract(T_N, a_P)
                 T_N1 ← extract(T_N, a_N)
                 T_P2 ← extract(T_N1, b_P)
                 T_N ← mix(T_P1, T_P2)
                 T_N ← purify(T_N)
          if v_F = c
          then { T_P1 ← extract(T_P, a_P)
                 T_N1 ← extract(T_P, a_N)
                 T_P2 ← extract(T_N1, b_P)
                 T_P ← mix(T_P1, T_P2)
                 T_P ← purify(T_P)
     T_P ← append(T_P, v_T)
     T_N ← append(T_N, v_F)
     T ← mix(T_P, T_N)
     T ← purify(T) }
return (detect(T))
```

Figure 3.4: OGIHARA AND RAY'S ALGORITHM evaluates each subsequent variable and determines possible assignments. The possible assignments for the variables a and b get extracted if c matches the current variable v. Effectively pruning only potential solutions. These potential solutions T_P and T_N get appended with the positive or negative string assignments. The algorithm continues until each variable gets evaluated. The remaining space T contains all solutions for the CNF instance ϕ after the algorithm terminates.

3.2.1 Description of Ogihara and Ray's algorithm

Ogihara and Ray's algorithm requires each input to have two attributes:

1. All clauses consist of exactly three literals.

2. All clauses must be ordered by the literal's variable.

Considering only 3-SAT instances fulfills Attribute (1). If models of k-SAT with $k > 3$ are desired as input, then a polynomial time reduction to 3-SAT must occur prior to execution.

Ordering the literals from each clause by variable fulfills Attribute (2). We impose an increasing ordering over the variable indices in each clause $C = (x_i \vee x_j \vee x_k)$. We have

$$1 \leq i < j < k \leq n.$$

We use the notation (a, b, c) to denote the ordered clause, where

$$a = x_i,$$

$$b = x_j,$$

$$c = x_k.$$

OGIHARA AND RAY'S ALGORITHM begins with four initial witness candidates.

$$T = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}$$

For example, let us evaluate the clause

$$C = x_1 \vee \neg x_2 \vee \neg x_3.$$

On the first iteration, we compare the third ordered literal c with x_3. Since $c = \neg x_3$, extract configurations that satisfy $a \vee b$.

$$T_P = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}$$

$$T_N = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}$$

From T, select configurations that satisfy $a_P = a_{\texttt{T}}$

$$T_{P1} = \{\texttt{STT}, \texttt{STF}\}.$$

From T, Select configurations that satisfy $a_N = a_{\texttt{F}}$

$$T_{N1} = \{\texttt{SFT}, \texttt{SFF}\}.$$

From T_{N1}, select configurations that satisfy $b_P = b_{\texttt{F}}$

$$T_{P2} = \{\texttt{SFF}\}.$$

Mix the contents of T_{P1} and T_{P2} as the contents of T_P

$$T_P = \{\texttt{STT}, \texttt{STF}, \texttt{SFF}\}.$$

We have the tubes

$$T_P = \{\texttt{STT}, \texttt{STF}, \texttt{SFF}\},$$

$$T_N = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}.$$

Finally append assignments that satisfy the current literal with T_P and T_F.

$$\begin{aligned} T_P &= \text{append}(T_P, c_{\texttt{F}}) \\ &= \{\texttt{STTF}, \texttt{STFF}, \texttt{SFFF}\} \end{aligned}$$

$$\begin{aligned} T_N &= \text{append}(T_N, c_{\texttt{T}}) \\ &= \{\texttt{STTT}, \texttt{STFT}, \texttt{SFTT}, \texttt{SFFT}\} \end{aligned}$$

Mix the contents of T_P and T_N to form the set of configurations that witness the clause.

$$T = \{\texttt{STTF}, \texttt{STFF}, \texttt{SFFF}, \texttt{STTT}, \texttt{STFT}, \texttt{SFTT}, \texttt{SFFT}\}$$

3.2.2 Detailed trace of Ogihara and Ray's algorithm

Appendix B lists a detailed execution trace for Ogihara and Ray's algorithm.

3.3 Implementations of molecular SATISFIABILITY solvers

We discuss existing implementations of molecular SATISFIABILITY solvers. Physical implementations apply molecular biology techniques and actual molecules. Simulation frame-

works use standard computation to simulate molecular biology techniques.

3.3.1 Physical implementations

Yoshida and Suyama implemented Ogihara and Ray's algorithm using manual molecular biology techniques [26]. This experiment solved a 3-CNF instance with four variables and 10 clauses.

Braich et al. implemented a molecular computer to filter solutions for a 3-SAT instance [3]. This experiment solved a 3-CNF instance with 20 variables and 24 clauses.

3.3.2 Simulation frameworks

Martn-Mateos et al. introduced a simulation for Lipton's algorithm [18]. Molecular operations get implemented in `ACL2`, a Common Lisp variant. The framework for this system implemented test cases for Lipton's algorithm.

Ogihara provides test results for an implementation of his original molecular algorithm [19]. This simulation provides a comparison to Lipton's algorithm for practical length restrictions.

Chapter 4

A new molecular algorithm for SATISFIABILITY

This chapter introduces a new molecular algorithm for SATISFIABILITY: the Distribution algorithm distributes literals from a CNF instance into a set of non-conflicting witnesses.

4.1 Distribution algorithm for SATISFIABILITY

The DISTRIBUTION ALGORITHM (Algorithm 4.1.1) starts with single literal witnesses from the first clause. The Distribution algorithm maintains a set of witnesses for each processed clause. Literal assignments that may be inserted into the witness candidates from the processed clauses we call *non-conflicting witnesses*. A witness candidate *conflicts* when it contains both x_i and $\neg x_i$ literal assignments. In this case, the algorithm removes the *conflicting witness* from the set of non-conflicting witnesses.

Algorithm 4.1.1: DISTRIBUTION ALGORITHM(ϕ)

for each clause C in ϕ

do $\begin{cases} T_C \leftarrow \emptyset \\ \textbf{for each} \text{ literal } \ell \text{ in } C \\ \quad \textbf{do} \begin{cases} T_I \leftarrow \text{INSERT LITERAL}(T, \ell) \\ T_C \leftarrow \text{mix}(T_C, T_I) \end{cases} \\ T \leftarrow \text{purify}(T_C) \end{cases}$

return ($detect(T)$)

Figure 4.1: DISTRIBUTION ALGORITHM constructs a set of non-conflicting assignments for a CNF instance. This algorithm inserts contents of each clause C_i into T with the INSERT LITERAL subroutine.

During execution, the Distribution algorithm maintains an ordering for literal assignments in a set of non-conflicting witness. The INSERT LITERAL subroutine (Algorithm 4.1.2) maintains an ordering to the literals in each ordered witness candidate.

Case 0 initiates an empty tube T with literal assignments from the first clause. We have the tube

$$\begin{aligned} T_0 &= \text{INSERT LITERAL}(T, x_1) \\ &= \{(x_1)\}. \end{aligned}$$

Let us consider an example to demonstrate the cases for INSERT LITERAL. Let $w = (x_2, \neg x_4, x_5)$ be a partial ordered witness contained in a tube

$$T = \{(x_2, \neg x_4, x_5)\}$$

Case 1:

$$\begin{aligned} T_1 &= \textsc{Insert Literal}(T, x_1) \\ &= \{(x_1) \cdot (x_2, \neg x_4, x_5)\} \\ &= \{(x_1, x_2, \neg x_4, x_5)\} \end{aligned}$$

Case 2:

$$\begin{aligned} T_2 &= \textsc{Insert Literal}(T, x_6) \\ &= \{(x_2, \neg x_4, x_5) \cdot (x_6)\} \\ &= \{(x_2, \neg x_4, x_5, x_6)\} \end{aligned}$$

Case 3:

$$\begin{aligned} T_3 &= \textsc{Insert Literal}(T, x_3) \\ &= \{(x_2) \cdot (x_3) \cdot (\neg x_4, x_5)\} \\ &= \{(x_2, x_3, \neg x_4, x_5)\} \end{aligned}$$

Case 4:

$$\begin{aligned} T_4 &= \textsc{Insert Literal}(T, \neg x_2) \\ &= \{(x_2, \neg x_4, x_5)\} \\ &= \{\} \end{aligned}$$

Case 5:

$$T_5 = \textsc{Insert Literal}(T, x_2)$$
$$= \{(x_2, \neg x_4, x_5)\}$$

In Case 1, the literal x_1 variable precedes the first literal in the ordered witness candidate w. In Case 2 the literal x_6 variable succeeds the last literal of the ordered witness candidate w.

Case 3 inserts a literal x_3 into w. Because the literal x_3 variable occurs between the first and last ordered literals, the ordered witness candidate w must be split to incorporate (x_3). We split w into two components

$$w_1 = (x_2)$$
$$w_2 = (\neg x_4, x_5),$$

and form the completed witness candiate by appending the literal x_3 to w_1

$$w_1 = (x_2, x_3).$$

Finally form the ordered non-conflicting witness w by appending w_2 to w_1

$$w = (x_2, x_3, \neg x_4, x_5).$$

Case 4 eliminates the conflicting witness; both positive (x_2) and negative $(\neg x_2)$ literals appear for the same variable in the witness candidate w.

In Case 5, the witness w remains unchanged since the literal x_2 exists in the witness candidate w.

Algorithm 4.1.2: INSERT LITERAL(T, ℓ)

$T_R \leftarrow \emptyset$
if $T = \emptyset$
  then { // Assign the single literal witness for **case** (0).
    $T_R \leftarrow \{(\ell)\}$
  else {
    for each Witness candidate w in T
      do {
        case (1) : $\ell < w$
          then $w' \leftarrow \text{append}(\ell, w)$
        case (2) : $\ell > s$
          then $w' \leftarrow \text{append}(w, \ell)$
        case (3) : $w_1 < \ell < w_2$
          then {
            $[w_1, w_2] \leftarrow \text{splice}(w, \ell)$
            $w_1 \leftarrow \text{append}(w_1, \ell)$
            $w' \leftarrow \text{append}(w_1, w_2)$
        case (4) : $\neg\ell \in w$
          then $w' \leftarrow \emptyset$
        case (5) : $\ell \in s$
          then $w' \leftarrow w$
        $T_R \leftarrow \text{mix}(T_R, w')$
return (T_R)

Figure 4.2: INSERT LITERAL maintains the literal assignment ℓ to a set of witness candidates contained in T. An ordering of literals is maintained for each oligonucleotide s. Cases (1), (2), and (3) insert the literal assignment ℓ into a witness configuration. Case (4) eliminates conflicting assignments those containing positive and negative literal assignments. Case (5) does not extend the witness contained in T if the literal assignment is redundant.

4.1.1 Description of the Distribution algorithm

The DISTRIBUTION ALGORITHM starts with the literal assignments of a clause. Evaluation of subsequent clauses extends the witness candidates using the INSERT LITERAL subroutine. Each insertion maintains the set of witness candidates to only those that satisfy the current clause. Table 4.1 lists the six possibilities for literal assignment.

Table 4.1: Configurations for the INSERT LITERAL subroutine

Case	**Ordered witness**	**State**
0	(ℓ)	if $T = \emptyset$
1	$(\ell) \cdot (w)$	if $v \in \ell$ is less than all literal indicies in w
2	$(w) \cdot (\ell)$	if $v \in \ell$ is greater than all literal indicies in w
3	$(w_1) \cdot (\ell) \cdot (w_2)$	if $v \in \ell$ is between two literal indicies in w
4	$\emptyset$	if ℓ conflicts with $\neg\ell$ in w
5	(w)	if ℓ exists in w

During this phase, each literal from a disjunctive clause incrementally constructs partial non-conflicting witnesses for the input ϕ. Case 0 inserts a literal ℓ to an unassigned initial tube. Cases 1, 2, and 3 place a literal ℓ into an expanding witness candidate w. Each of these cases maintain an ordering to the expanding set of non-conflicting witness candidates.

A literal conflict occurs when both positive and negative assignments of a literal occur in a witness candidate w. Case 4 removes the conflicting witness candidate w from the set potential solutions. In Case 5, the witness w remains unmodified; this case occurs when the witness w contains the literal ℓ.

For example, let us consider the CNF instance

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3).$$

The Distribution algorithm begins with the first clause. We have the set of witnesses

$$T = \{(x_1), (x_2)\}.$$

Next, insert the literals from the second clause into the set of non-conflicting witnesses in T.

$$T_C = \emptyset$$

First, insert x_1 into T

$$T_I = \{(x_1), (x_1, x_2)\}.$$

The witness (x_1) contains x_1, and the idempotent witness (x_1) remains unchanged. The witness (x_1, x_2) requires x_1 to satisfy the second clause.

Mix the contents of T_I into the set of clause witnesses T_C

$$T_C = \{(x_1), (x_1, x_2)\}.$$

Next, insert $\neg x_2$ into T

$$T_I = \{(x_1, \neg x_2)\}.$$

The witness $(x_1, \neg x_2)$ requires $\neg x_2$ to satisfy the second clause. The witness candidate $(x_2, \neg x_2)$ contains a conflict and gets removed from the set of potential witnesses.

Mix the contents of T_I into the set of clause witnesses T_C

$$T_C = \{(x_1), (x_1, x_2), (x_1, \neg x_2)\}.$$

Finally, insert the literal x_3 into T

$$T_I = \{(x_1, x_3), (x_2, x_3)\}.$$

In this case, the literal x_3 is required to satisfy the second clause. The literal gets inserted into the configurations without redundant or conflicting assignment.

Mix the contents of T_I into the set of clause witnesses T_C

$$T_C = \{(x_1), (x_1, x_2), (x_1, \neg x_2), (x_1, x_3), (x_2, x_3)\}.$$

Assign the contents of T_C to T

$$T = \{(x_1), (x_1, x_2), (x_1, \neg x_2), (x_1, x_3), (x_2, x_3)\}.$$

The set T contains non-conflicting witnesses for the observed clauses. Once the algorithm terminates, T contains non-conflicting witnesses for the CNF instance ϕ.

4.1.2 Detailed trace of the Distribution algorithm

Appendix B lists a detailed execution trace for the Distribution algorithm.

Chapter 5

Molecular Simulation: A system for molecular computation

This chapter introduces Molecular Simulation: A system for molecular computation. We describe an overview of Molecular Simulation and its documentation, along with tools for automated execution for Molecular Simulation. This includes `Perl` execution scripts and visualization for output data. We describe example input and output for Molecular Simulation. Command line argument provides user configurable options for Molecular Simulation. Chapter 6 describes the usage of Molecular Simulation with automated execution.

5.1 Overview

Molecular Simulation implements a simulated molecular lab for operating on DNA. The present simulation implements three molecular algorithms for SATISFIABILITY. The included `Perl` scripts process DIMACS CNF input directories with invocations to Molecular Simulation.

Molecular Simulation may be executed directly or invoked with the assistance of an

execution script. The system requirements to execute or design a molecular experiment are listed in this section.

This program is a simulated molecular lab for experimenting with DNA operations. Implementation of three molecular algorithms for solving SATISFIABILITY include Lipton's algorithm, Ogihara and Ray's algorithm, and the Distribution algorithm. Chapters 3 and 4 describe the background and provide pseudocode for these algorithms.

5.2 Download

Download Molecular Simulation from: `https://github.com/dncarley/MolecularSimulation`.

5.3 Requirements

This section specifies the requirements for running Molecular Simulation.

5.3.1 Hardware requirements

Molecular Simulation requires a 64-bit processor with 2 GB of RAM.

5.3.2 Software requirements

`gcc` (GNU Compiler Collection) must be installed to build Molecular Simulation.

`Perl` must be installed to automate build and execution of Molecular Simulation.

5.4 Documentation

The project website contains detailed documentation for Molecular Simulation. The documentation provides an overview of Molecular Simulation that may be used independently

of Chapters 5 and 6 for getting started. The online documentation describes detailed datatype, function, and class definitions.

5.5 Tools

This project uses several tools for automating tasks and execution. In this section, we discuss tools to automate execution and visualize output from Molecular Simulation.

5.5.1 `Perl` utilities

The source directory includes several `Perl` scripts to assist in building and initiation of tests for Molecular Simulation. Table 5.1 documents the basic usage for build and testbench execution scripts. Each script provides detailed execution options.

5.5.2 Data Visualization

Ben Fry's example in Chapter 4 of *Visualizing Data* [8] provides a framework for importing output from Molecular Simulation.

The SAT Datapoint Visualization for Molecular Simulation's output can be downloaded from: `https://github.com/dncarley/VisualizeSatDatapoints`.

5.6 Input

Input to Molecular Simulation consists of a DIMACS CNF file. The definition of the `*.cnf` filetype can be accessed from: `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/`.

```
c comments begin with a 'c'
```

Table 5.1: `Perl` execution commands and descriptions.

`Perl` **script**	**Usage**	**Description**
`build.pl`	`$ perl build.pl`	Compiles Molecular Simulation and generates an executable in the directory `./execute/simulation`.
`buildGenerate.pl`	`$ perl buildGenerate.pl`	Generates a sweep of CNF formulas over a range of k-SAT ratios. Program uses a modified random k-SAT generator from Microsoft Research.
`executeMolecularSat.pl`	`$ perl executeMolecularSat.pl`	Executes Molecular Simulation for a directory of SATISFIABILITY instances with desired algorithms. If no options are specified, then each of the three algorithms are executed and output is generated in the same test directory.
`runSimulation.pl`	`$ perl runSimulation.pl`	Executes `build.pl` followed by `executeMolecularSat.pl`. Any command line arguments get passed to `executeMolecularSat.pl`

```
c
c cnf input is designated with ‘p cnf’
c    followed by number of variables <n>, and clauses <m>
c
p cnf 4 3
c
c A clause is represented by a sequence of <k> integers,
c     separated by whitespace and ending with a ‘0’.
c Each variable is represented by the integer sequence,
c    negative polarity is represented by ‘-’.
c
1 2 -3 0
2 3 -4 0
-1 -3 -4 0
```

5.7 Output

Output from Molecular Simulation, by default, conforms to the 2011 SAT Competition rules. The rules can be accessed from: `http://www.satcompetition.org/2011/rules.pdf`.

```
c comments begin with a ‘c’
c
s SATISFIABLE
c
c A line beginning with a ‘s’ marks the status.
```

```
c This can be either 'UNSATISFIABLE', 'SATISFIABLE', or 'UNKNOWN'.
c
v 1 2 -3 -4 0
c
c A satisfiable witness begins with a 'v' and ends with a '0'.
c     A sequence of integers, between 'v' and '0', encodes a satisfiable assignment.
```

Table 5.2 describes an extended custom output. This output reports parameters for metric performance evaluation.

Table 5.2: Molecular Simulation output logging.

Parameter	**Description**
`c algorithmType:`	Display the algorithm type: `Lipton`, `Ogihara-Ray`, `Distribution`
`c algorithmTime:`	Display the algorithm execution time in seconds.
`c solutionMemory:`	Display the solution space memory in Bytes.
`c mixCount:`	Display the number of `mixes` required during algorithm execution.
`c extractCount:`	Display the number of `extracts` required during algorithm execution.
`c appendCount:`	Display the number of `appends` required during algorithm execution.
`c splitCount:`	Display the number of `splits` required during algorithm execution.
`c spliceCount:`	Display the number of `splices` required during algorithm execution.
`c purifyCount:`	Display the number of `purifications` required during algorithm execution.
`c numVar:`	Display the number of `variables` in the input CNF instance.
`c numClause:`	Display the number of `clauses` in the input CNF instance.

5.8 Execution

Invocation of Molecular Simulation can be performed from the command line.

```
$ ./execute/simulation i [input] [options]
```

The `[input]` consists of a DIMACS CNF file. Command line `[options]` may be a combination of the options in Table 5.3.

Table 5.3: Command line options for Molecular Simulation

Argument	Parameters	Description
`-a`		Algorithm select
	`d`	Distribution algorithm
	`l`	Lipton's algorithm
	`o`	Ogihara and Ray's algorithm
`-d`		Debug
`i`		Input
	`[input]`	DIMACS CNF file
`-w`		Write output to file
	`[output]`	Output filename

5.8.1 Execution example

Suppose that we would like to execute Ogihara and Ray's algorithm for a DIMACS CNF file instance `test1.cnf` located in the directory `MolecularSimulation/testbench`. We output the results `test1-o.out` in the same directory as the input CNF.

We invoke Molecular Simulation with the following command:

```
$ ./execute/simulation i ../testbench/test1.cnf -a o -w ../testbench/test1-o.out
```

In the next chapter, we will describe the automation for a random k-SAT sweep with each of the algorithms. The provided `Perl` scripts are the recommended method for build-

ing and execution of Molecular Simulation.

Chapter 6

Experimental Setup

This chapter describes the use of Molecular Simulation for evaluation of a set of DIMACS CNF Satisfiability instances. We discuss configuration for generation of random k-Sat instances. Further, any existing DIMACS CNF benchmark may be imported for test. Example configuration options automate the execution of Molecular Simulation. The example continues with an analysis of runtime metrics for each test instance. The next chapter describes the results from the k-Sat sweep experiment.

6.1 Setup

In this section, we describe prerequisites for executing a test bench using Molecular Simulation. Molecular Simulation requires a 64-bit architecture with a UNIX like system with `gcc` and `Perl`. The target system must meet the minimum requirements.

Building Molecular Simulation can be performed by invoking the `Perl` script `build.pl` from the command line.

```
$ perl build.pl
```

This script generates an executable `simulation` in the directory:

```
MolecularSimulation/execute
```

The next sections describe invocation of Molecular Simulation with desired options. We begin with the creation and importation of DIMACS CNF datasets.

6.2 Create dataset

We create a sweep of random k-SAT instances to observe SAT phase transition. David Wilson's `ksat.c` generates random k-SAT instances in DIMACS CNF format [25]. The program takes four arguments to create a unique DIMACS CNF instance. Invocation of the program can be performed using the following command.

```
$ ./execute/ksat k n m s > output.cnf
```

This generates *output*`.cnf` in DIMACS CNF format with k variables per clause n variables, m clauses, and random seed s.

We use automated `Perl` scripts to create a sweep of DIMACS CNF instances. Setup for a sweep configuration includes specifying a set of ratios. Invocation of the script generates a set of random k-SAT instances. The redirected output gets stored in the target directory with the previous file naming convention. We use the following command to invoke the construction of a sweep of k-SAT instances.

```
$ perl buildGenerate.pl
```

6.3 Import dataset

Datasets of DIMACS CNF input may be provided for batch processing. This includes random k-SAT instances generated from the previous section, or importing existing DIMACS

CNF instances.

DIMACS CNF benchmarks are available for download from: `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`.

6.4 Configure test

The previous chapter described a single execution of Molecular Simulation. We use automated scripts for processing datasets with each of the algorithms.

The `Perl` script `executeMolecularSat.pl` allows execution for a directory of DIMACS CNF input. Executing the script from the command line without arguments processes the experimental setup and saves output to the same directory.

```
$ perl executeMolecularSat.pl [options]
```

The options for `executeMolecularSat.pl` can be a combination of the options in Table 6.1.

Table 6.1: Command line options for `executeMolecularSat.pl`

Argument	Parameters	Description

`-d`
`-l`
`-o`
		Distribution algorithm
		Lipton's algorithm
		Ogihara and Ray's algorithm
		
		Default: Execute all three algorithms.

`-debug`		Debug
`-p`	`[CNF file path]`	Specify CNF file path. **Default path:** `data/testCNF`
`-f`		Write output to file

6.5 Execution and collection of data

The following command builds and executes Molecular Simulation.

```
$ perl runSimulation.pl [options]
```

This command first builds Molecular Simulation with `build.pl`, and invokes Molecular Simulation with `executeMolecularSat.pl`. The `[options]` are passed directly to `executeMolecularSat.pl`. Molecular Simulation executes the default experimental setup with no `[options]` specified.

Output consists of the standard SAT Competition output appended with custom runtime metric logging. Collections of output files may be read by the data visualization program and exported into a condensed table.

6.5.1 Execution output

Molecular Simulation, by default, writes output to standard output on the console. The `-f` option saves output to a file as `[filename]-<a>.out`. The `[filename]` consists of the DIMACS CNF name and `<a>` specifies the algorithm type: `d`, `l` or `o`.

Output directed to standard output conforms to the SAT Competition rules. This output may be used during testing, or redirected to an external stream. The debug option `-debug` displays detailed information about the execution. The debug option writes verbose content based on the program execution.

Reading output metrics from the saved output, as defined in Table 5.2, allows for analysis of collected data. The data visualization reads a directory of output and condenses it as a Tab Separated Values (`*.tsv`) file. Subsequent datapoint browsing and the online view use the `*.tsv` file for condensed reading and transmission. In the next chapter, we describe the results of the experimental setup.

Chapter 7

Results

This chapter presents results of the k-SAT execution test from the previous chapter. We consider the results of the test and analyze the algorithm metrics.

7.1 Algorithm metric comparison

This section describes the results from the simulation. We analyze the molecular operations count for append, extract, mix, purify, splice, and split. Presentation of actual computation time and required memory for the solution representation allow for comparison of algorithms.

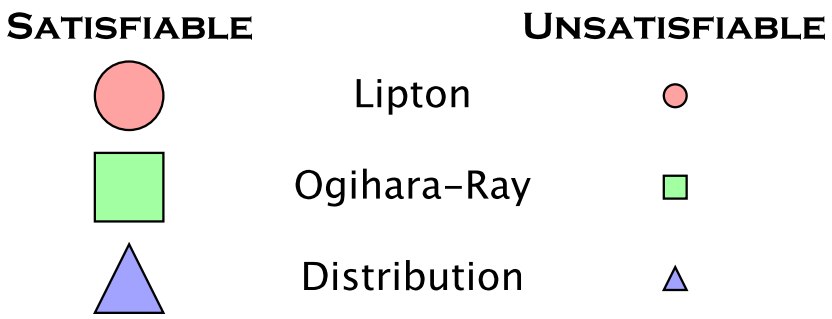

Figure 7.1: Key for output metrics. Large shapes represent satisfiable instances and small shapes represent unsatisfiable instances. Datapoints for Lipton's algorithm are represented with red circles, Ogihara and Ray's algorithm with green squares, and the Distribution algorithm with blue triangles.

Append concatenates two oligonucleotides.

The Distribution algorithm is exponential in the number of appends. The operation count for append depends on the parsing order of the CNF instance.

Lipton's and Ogihara-Ray's algorithms use a fixed number of appends. This depends on the number of variables and clauses present in the CNF instance.

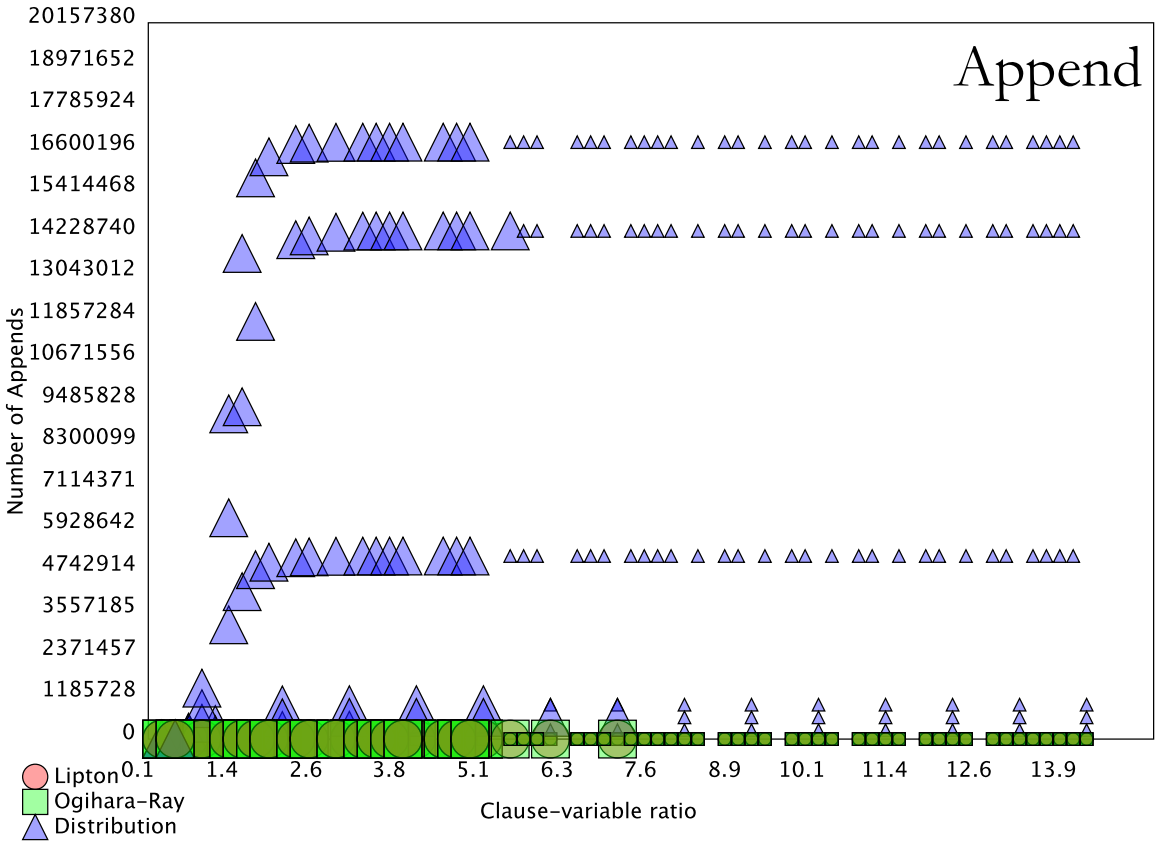

Figure 7.2: Clause to variable ratio α vs. Number of appends

Extract filters oligonucleotides from a tube.

Ogihara-Ray's algorithm requires the greatest number of extracts. Lipton's algorithm is linear on α and varies a constant from Ogihara-Ray's algorithm.

The Distribution algorithm does not require extract.

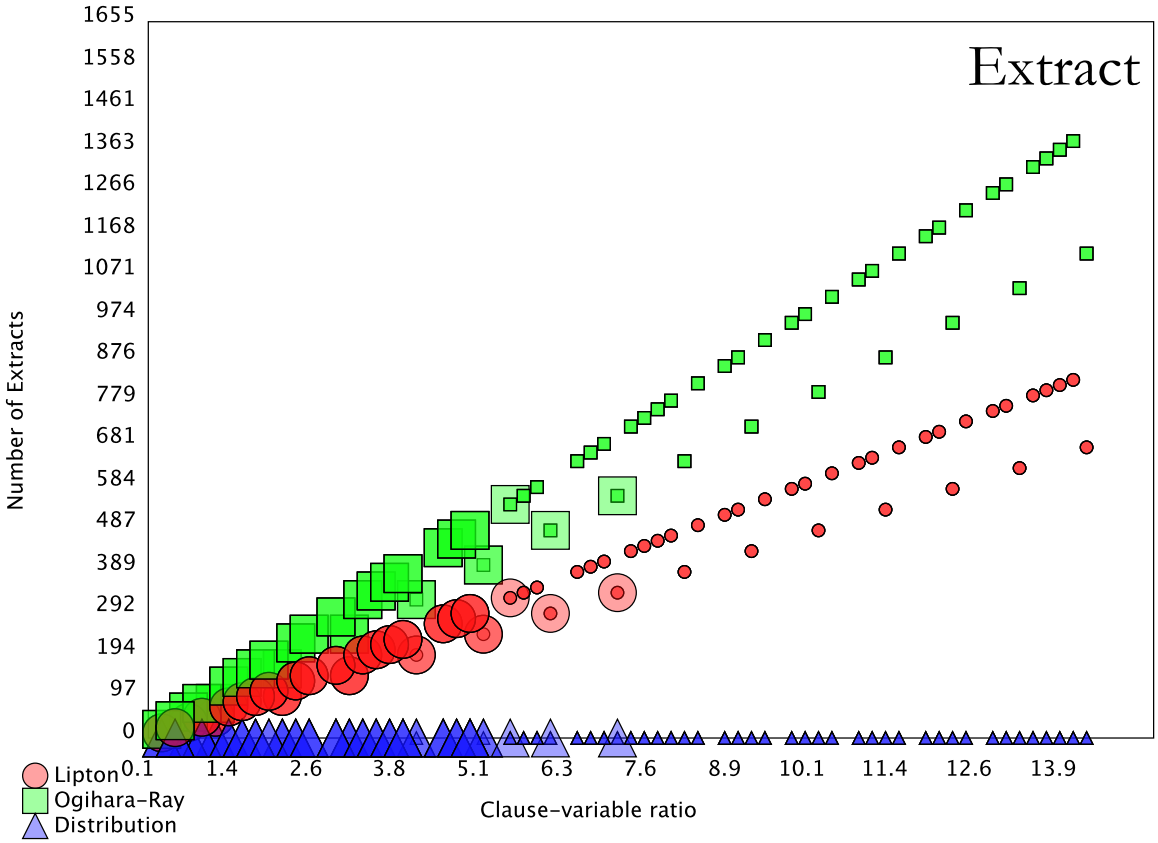

Figure 7.3: Clause to variable ratio α vs. Number of extracts

Mix combines the contents of two tubes.

Lipton's algorithm requires a linear number of mixes on α. The Distribution algorithm also requires a linear number of mixes, varying by a constant factor from Lipton's algorithm.

Ogihara-Ray's algorithm requires a constant number of mixes on α.

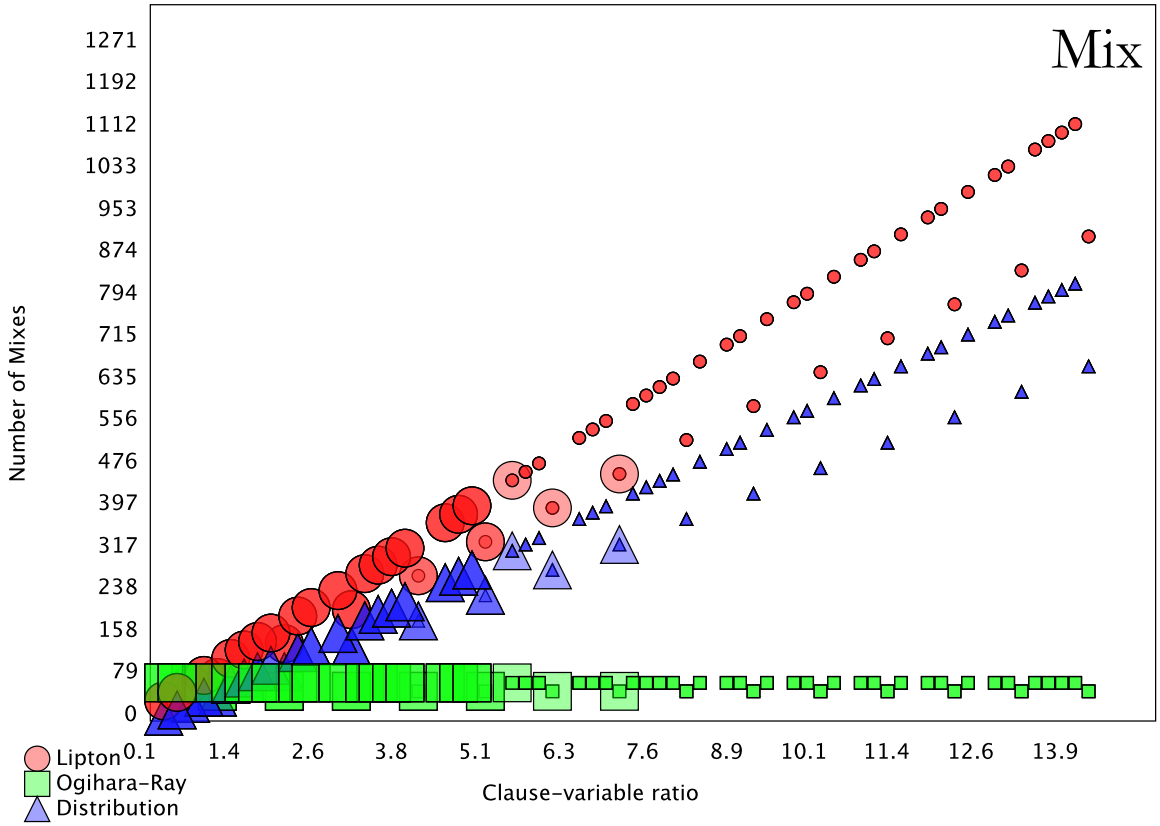

Figure 7.4: Clause to variable ratio α vs. Number of mixes

Purify ensures a uniform distribution of each independent oligonucleotide in a tube.

All three algorithms operate using a linear number of purifications on α. Ogihara-Ray's algorithm requires the greatest number of purifications. The purifications vary by a constant when compared to Lipton's and the Distribution algorithms.

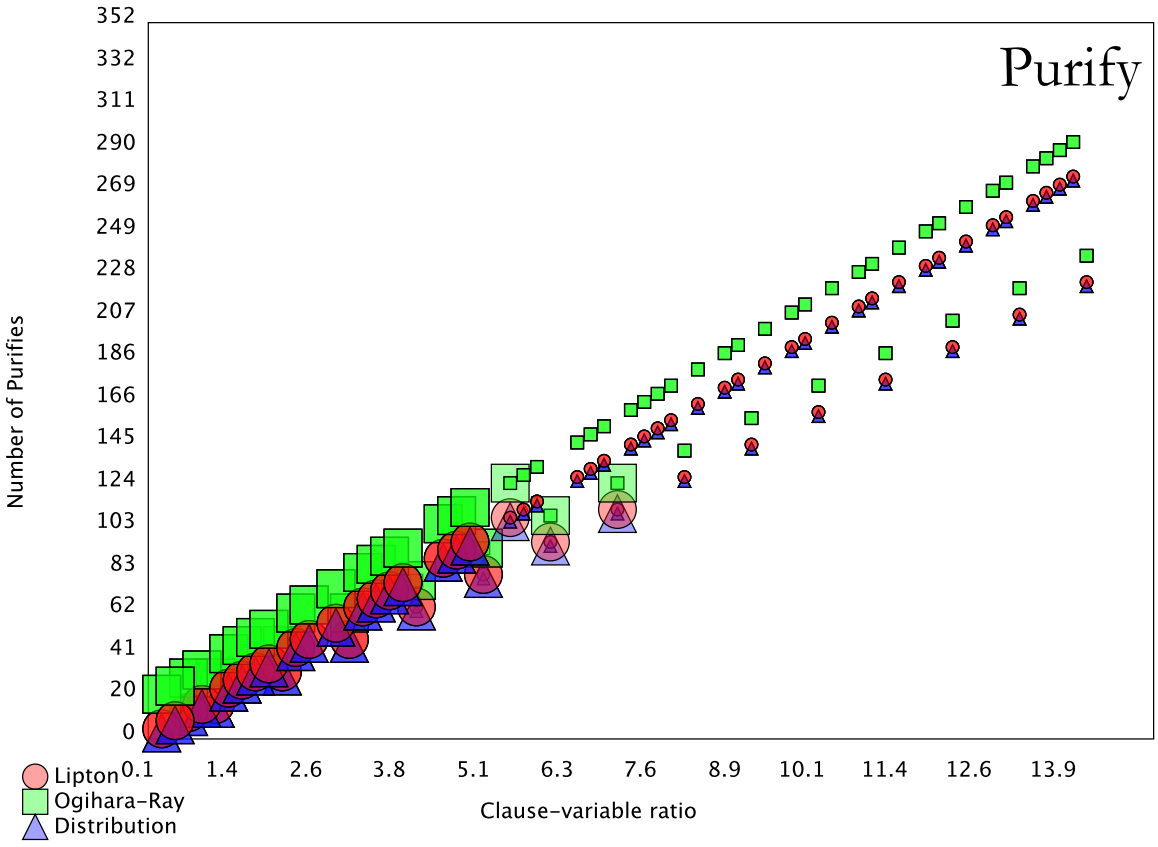

Figure 7.5: Clause to variable ratio α vs. Number of purifies

Splice cuts an oligonucleotide at a targeted location.

The Distribution algorithm is exponential in the number of splices. The number of splices depends on the parsing order of the CNF instance. Each split requires reassembly, accomplished using two appends. Figure 7.2 shows the number of appends.

Lipton's and Ogihara-Ray's algorithms do not require the splice operator.

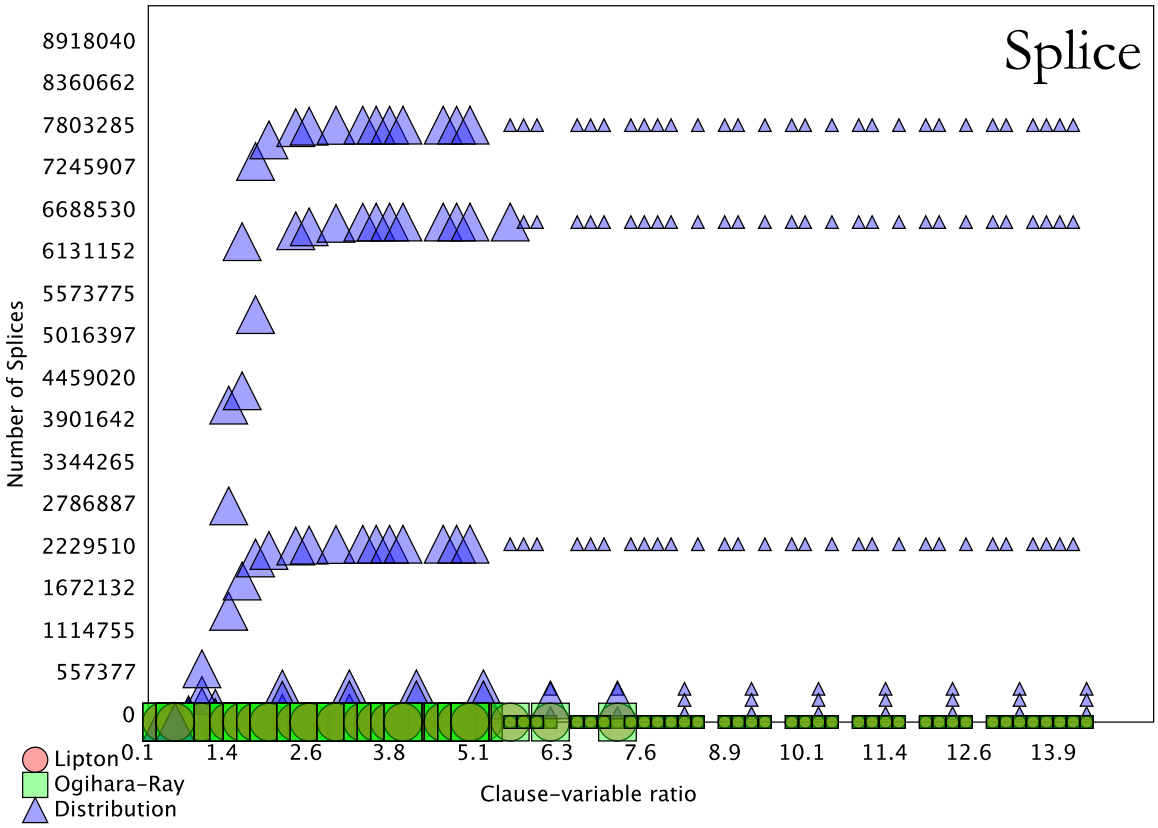

Figure 7.6: Clause to variable ratio α vs. Number of splices

Split portions a tube into two exact tubes.

The Distribution algorithm requires a linear number of splits.

Lipton's and Ogihara-Ray's algorithms are constant in splits based the number of variables.

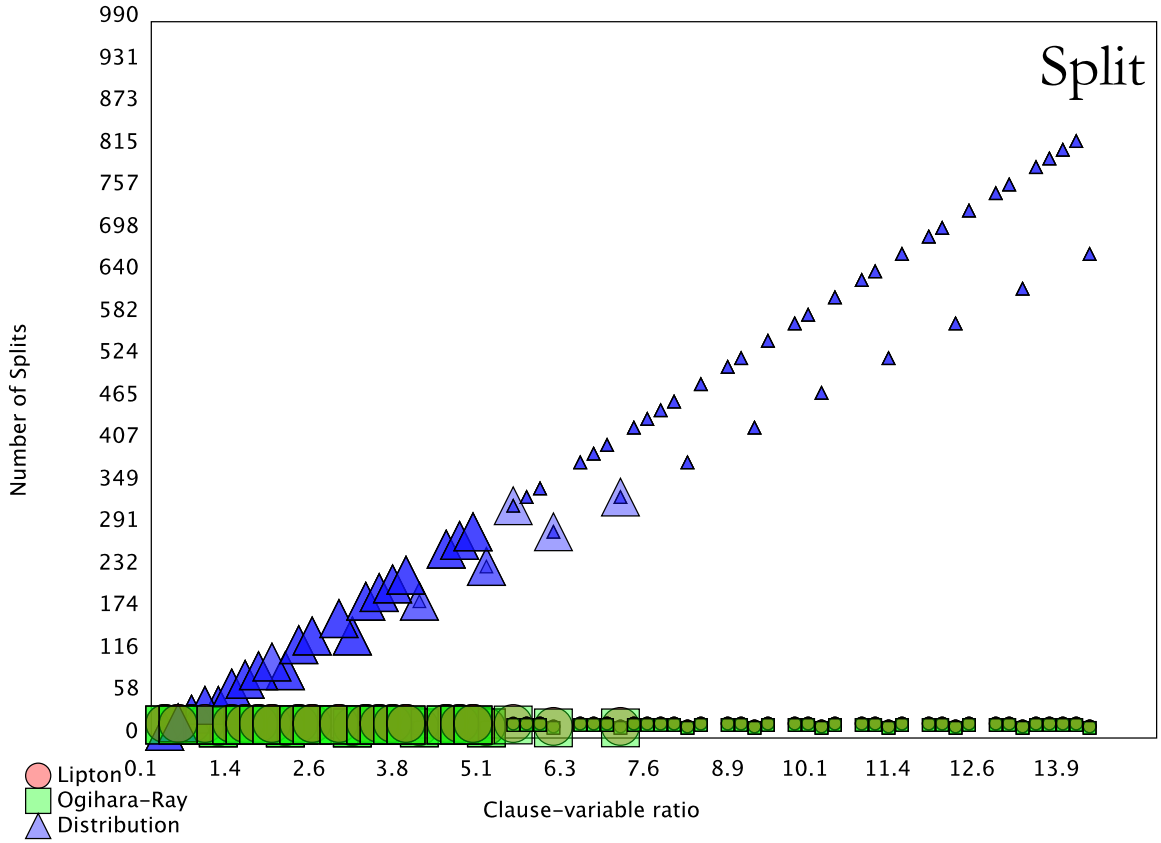

Figure 7.7: Clause to variable ratio α vs. Number of splits

Time measures algorithm execution time in seconds.

Ogihara-Ray's algorithm requires the least time. In cases where the SATISFIABILITY instance is under-constrained, where more possible solutions occur, the algorithm takes the greatest time. Less pruning occurs in over-constrained instances, reducing the execution time of test instances.

Lipton's algorithm executes in exponential time $\alpha \approx [4.2, 8.2]$ (the phase transition region for 3-SAT) taking the longest.

The Distribution algorithm executes in exponential time, and performs better than Lipton's algorithm for low conflict ratios. However over the entire sweep performs worse than both Lipton's and Ogihara-Ray's algorithms. It shares the same $\alpha \approx [4.2, 8.2]$ during the 3-SAT phase-transition.

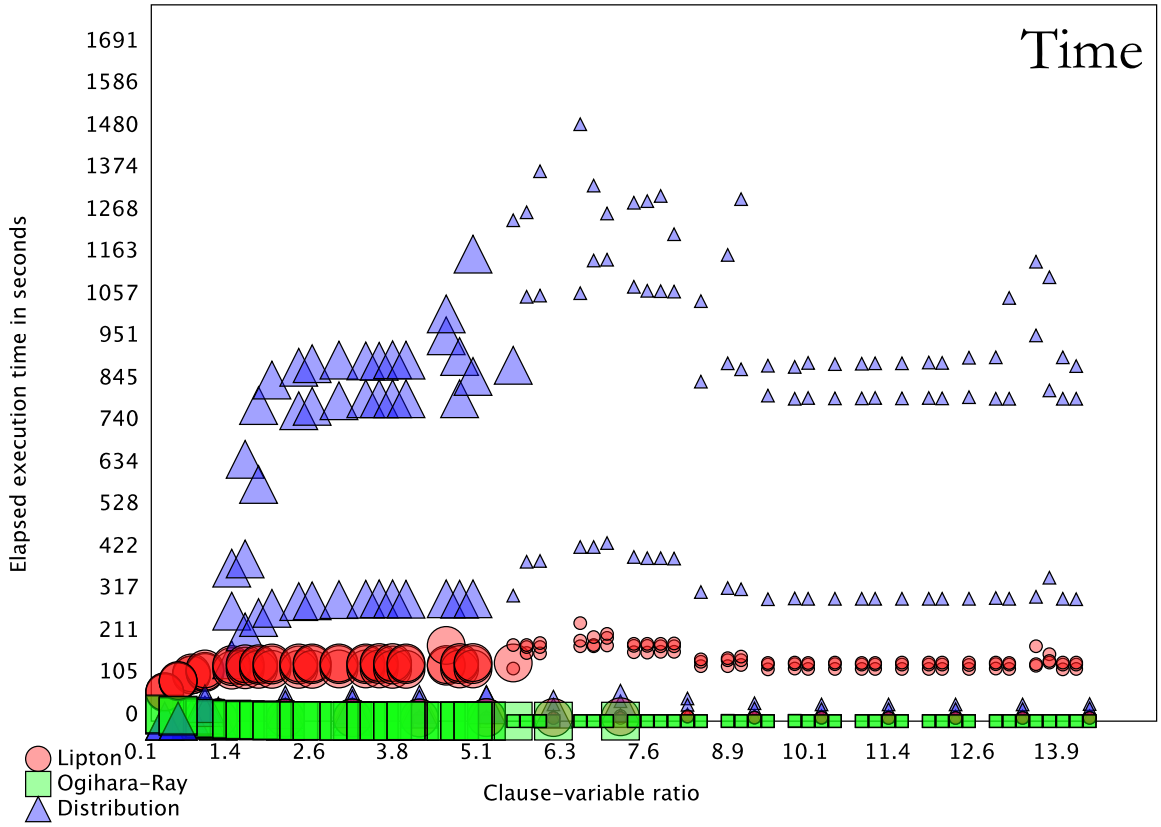

Figure 7.8: Clause to variable ratio α vs. execution time in seconds

Memory measures witness footprint in Bytes.

Lipton's and Ogihara-Ray's algorithms share the same solution footprint.

The Distribution algorithm contains a larger solution footprint after the trivially satisfiable instances with $\alpha \approx [0.2, 0.8]$. The space contains a set of non-conflicting assignments from $\alpha \approx [0.8, 2.9]$. Non-conflicting assignments consist of witnesses for only necessary literals.

Each SATISFIABILITY instance has a constrained solution space during the phase-transition region. All three algorithms share the same footprint. There are no satisfiable instances in this test with $\alpha > 7.2$. The axis in Figure 7.9 scales accordingly.

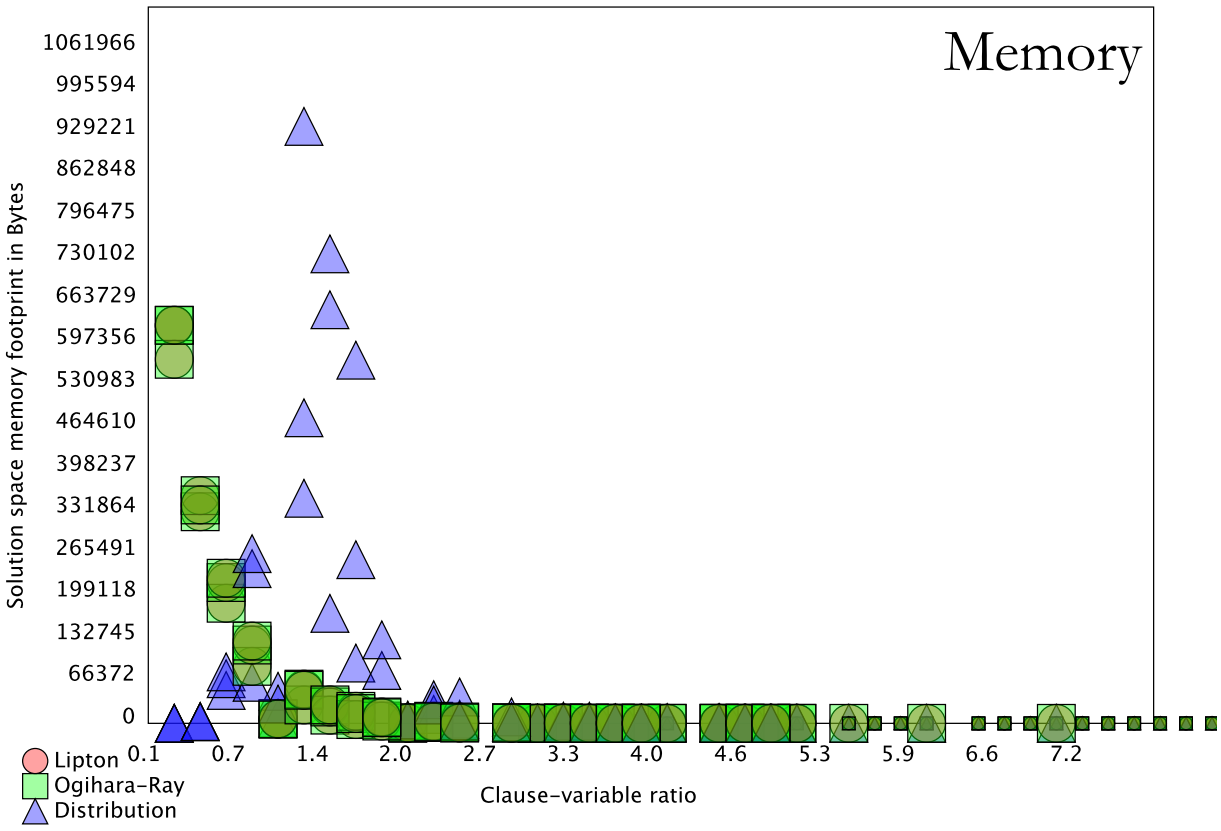

Figure 7.9: Clause to variable ratio α vs. satisfiable solution footprint in Bytes

Chapter 8

Conclusions

This project considered SATISFIABILITY as a problem for general computation. We considered three molecular algorithms for SATISFIABILITY and simulated their execution with a computation framework. In this chapter, we state the contributions of this project and directions molecular computation will take.

8.1 Contributions

We developed several contributions for molecular computation during this project. This includes introducing the Distribution algorithm for SATISFIABILITY in Chapter 4. We introduced Molecular Simulation in Chapter 5 and collected data from simulations of three molecular SATISFIABILITY algorithms described in Chapter 6. This comparison shows advantages and disadvantages of each of the molecular algorithms for SATISFIABILITY.

8.2 Future work

Gene sequencers have been designed for reading molecules and diagnosing patients in a medical setting. These gene sequencers currently have the ability to sequence any type of gene. Observing real interactions between sequences in a controlled environment permit molecular computation for targeted gene disease diagnosis.

SATISFIABILITY defines a canonical input format for combinatorial problems. Applications of molecular SATISFIABILITY algorithms may extend to witness natural gene expressions. Gene sequencers designed for molecular computation permit observation of molecular interactions on both synthetic and natural gene expressions.

Bibliography

[1] Adleman, L. M. Molecular computation of solutions to combinatorial problems. *Science 266* (November 1994), 1021–1024.

[2] Baltimore, D. Expression of animal virus genomes. *Bacteriol Rev 35*, 3 (1971), 235–41.

[3] Braich, R. S., Chelyapov, N., Johnson, C., Rothemund, P. W. K., and Adleman, L. Solution of a 20-variable 3-sat problem on a DNA computer. *Science 296* (2002), 499–502.

[4] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), STOC '71, ACM, pp. 151–158.

[5] DIMACS. Satisfiability suggested format. Accessed from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`. DIMACS 1993. Accessed from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`. DIMACS 1993., May 1993.

[6] Doherty, P., and Kvarnström, J. *The Handbook of Knowledge Representation*. Elsevier, 2008.

[7] Feynman, R. There's Plenty of Room at The Bottom. Accessed from: `http://resolver.caltech.edu/CaltechES:23.5.0`. *Caltech Engineering and Science 23*, 5 (1960).

[8] Fry, B. *Visualizing Data*. O'Reilly Media Inc., 2008.

[9] Furka, A. Study on possibilities of systematic searching for pharmaceutically useful peptides. *Notarized on May 29, 1982. Accessed from* `http://szerves.chem.elte.hu/furka/` (May 1982).

[10] Furka, A. *Combinatorial Chemistry Combinatorial Chemistry Principles and Techniques*. -, 2007.

[11] Garaj, S., Hubbard, W., Reina, A., Kong, J., Branton, D., and Golovchenko, J. A. Graphene as a subnanometre trans-electrode membrane. *Nature 467*, 7312 (Sept. 2010), 190–193.

[12] Gent, I. P., and Walsh, T. The SAT phase transition. In *ECAI* (1994), John Wiley & Sons, pp. 105–109.

[13] Ignatova, Z., Martinez-Perez, I., and Zimmerman, K.-H. *DNA Computing Models*. Springer, 2008.

[14] Levin, L. Universal search problems (in Russian). *Problemy Peredachi Informatsii 9*, 3 (1973), 115–116.

[15] Life Technologies. Ion Torrent. Accessed from `http://www.iontorrent.com/`.

[16] Lipton, R. Using DNA to solve NP-complete problems. *Science 268* (1995), 542–545.

[17] Loughran, M. IBM Research Aims to Build Nanoscale DNA Sequencer to Help Drive Down Cost of Personalized Genetic Analysis. Accessed from: http://www-03.ibm.com/press/us/en/pressrelease/28558.wss, October 2009.

[18] Martín-Mateos, F., Alonso, J. A., Perez-Jimenez, M., and Sancho-Caparrini, F. Molecular computation models in ACL2: a simulation of Lipton's experiment solving sat, 2002.

[19] Ogihara, M. Breadth first search 3-sat algorithms for DNA computers. Tech. rep., University of Rochester, Rochester, NY, USA, 1996.

[20] Ogihara, M., and Ray, A. DNA-based parallel computation by "counting". Tech. rep., University of Rochester, 1997.

[21] Oxford Nanopore Technologies. Oxford Nanopore Technologies. Accessed from http://www.nanoporetech.com/.

[22] SATComp Organizing committee. The international sat Competitions web page. Accessed from http://satcompetition.org/.

[23] Sipser, M. *Introduction to the Theory of Computation, Second Edition.* Course Technology, 2006.

[24] Stankovich, S., Dikin, D. A., Dommett, G. H. B., Kohlhaas, K. M., Zimney, E. J., Stach, E. A., Piner, R. D., Nguyen, S. T., and Ruoff, R. S. Graphene-based composite materials. *Nature 442*, 7100 (2006), 282–6.

[25] Wilson, D. Random k-sat generator. *Accessed from: http://research.microsoft.com/en-us/um/people/dbwilson/ksat/default.htm* (2011).

[26] Yoshida, H., and Suyama, A. Solution to 3-Sat by breadth first search. In *DNA Based Computers V* (2000), E. Winfree and D. Gifford, Eds., vol. 54 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pp. 9–22.

[27] Zhang, W. Phase transitions and backbones of 3-sat and maximum 3-sat. In *Principles and Practice of Constraint Programming — CP 2001*, T. Walsh, Ed., vol. 2239 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 153–167.

Appendix A

Source

A.1 Contributed

Download Molecular Simulation:

- `https://github.com/dncarley/MolecularSimulation`
- Documentation
  - Online Documentation:
    - `http://www.cs.rit.edu/~dnc6813/project/generatedDocs/index.html`
  - Offline Documentation:
    - `http://www.cs.rit.edu/~dnc6813/project/refman.pdf`

Download Sat Datapoints Visualization:

- `https://github.com/dncarley/VisualizeSatDatapoints`

A.2 External

Download David Wilson's k-SAT Generator:

- `http://research.microsoft.com/en-us/um/people/dbwilson/ksat/default.htm`

Download Doxygen:

- `http://www.stack.nl/~dimitri/doxygen/`

Download Ben Fry's examples for *Visualizing Data*:

- `http://benfry.com/writing/archives/3`

Appendix B

Molecular algorithm trace

B.1 Example SATISFIABILITY instance

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

B.2 Lipton's Algorithm

$$T = \text{COMBINATORIAL GENERATE}(4)$$

$$T =$$

```
STTTT SFTTT STFTT SFFTT STTFT SFTFT STFFT SFFFT
STTTF SFTTF STFTF SFFTF STTFF SFTFF STFFF SFFFF
```

Next select Clause 1:

$$C_1 = (x_1 \vee x_2 \vee \neg x_3)$$

STTTT SFTTT STFTT SFFTT STTFT SFTFT STFFT SFFFT

STTTF SFTTF STFTF SFFTF STTFF SFTFF STFFF SFFFF

Extract x_1:

STTTT STFTT STTFT STFFT

STTTF STFTF STTFF STFFF

Extract x_2:

STTTT SFTTT STTFT SFTFT

STTTF SFTTF STTFF SFTFF

Extract $\neg x_3$:

STTFT SFTFT STFFT SFFFT

STTFF SFTFF STFFF SFFFF

Mix contents:

STTTT SFTTT STFTT STTFT SFTFT STFFT SFFFT

STTTF SFTTF STFTF STTFF SFTFF STFFF SFFFF

Next select Clause 2:

$$C_2 = (x_2 \vee x_3 \vee \neg x_4)$$

STTTT SFTTT STFTT STTFT SFTFT STFFT SFFFT

STTTF SFTTF STFTF STTFF SFTFF STFFF SFFFF

Extract x_2:

```
STTTT SFTTT          STTFT SFTFT
STTTF SFTTF          STTFF SFTFF
```

Extract x_3:

```
STTTT SFTTT STFTT
STTTF SFTTF STFTF
```

Extract $\neg x_4$:

```
STTTF SFTTF STFTF      STTFF SFTFF STFFF SFFFF
```

Mix contents:

```
STTTT SFTTT STFTT      STTFT SFTFT
STTTF SFTTF STFTF      STTFF SFTFF STFFF SFFFF
```

Finally, select Clause 3:

$$C_3 = (\neg x_1 \vee \neg x_3 \vee x_4)$$

```
STTTT SFTTT STFTT      STTFT SFTFT
STTTF SFTTF STFTF      STTFF SFTFF STFFF SFFFF
```

Extract $\neg x_1$:

```
  SFTTT                SFTFT
  SFTTF                SFTFF      SFFFF
```

Extract $\neg x_3$:

```
                  STTFT SFTFT
                  STTFF SFTFF STFFF SFFFF
```

Extract x_4:

```
STTTT SFTTT STFTT      STTFT SFTFT
```

Mix contents:

```
STTTT SFTTT STFTT      STTFT SFTFT
      SFTTF            STTFF SFTFF STFFF SFFFF
```

B.3 Ogihara and Ray's Algorithm

Initialize the tube T with initial vector assignments for variables x_1 and x_2

$$T = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}$$

Iterate variable x_3:

$$C_1 = (x_1 \vee x_2 \vee \neg x_3)$$

$\neg x_3$ matches v_3

$$T_{P1} = \{\texttt{STT}, \texttt{STF}\}$$

$$T_{N1} = \{\texttt{SFT}, \texttt{SFF}\}$$

$$T_{P2} = \{\texttt{SFT}\}$$

$$T_P = \{\texttt{STT}, \texttt{STF}, \texttt{SFT}\}$$

$$C_2 = (x_2 \vee x_3 \vee \neg x_4)$$

x_3 or $\neg x_3$ does not match v_3

$$C_3 = (\neg x_1 \vee \neg x_3 \vee x_4)$$

x_3 or $\neg x_3$ does not match v_3

Append

$$T_P = \{\texttt{STTT}, \texttt{STFT}, \texttt{SFTT}\}$$

$$T_N = \{\texttt{STTF}, \texttt{STFF}, \texttt{SFTF}, \texttt{SFFF}\}$$

Mix

$$T = \{\texttt{STTT}, \texttt{STFT}, \texttt{SFTT}, \texttt{STTF}, \texttt{STFF}, \texttt{SFTF}, \texttt{SFFF}\}$$

Iterate variable x_4:

$$C_1 = (x_1 \vee x_2 \vee \neg x_3)$$

x_4 or $\neg x_4$ does not match v_3

$$C_2 = (x_2 \vee x_3 \vee \neg x_4)$$

$\neg x_4$ matchs v_3

$$T_{P1} = \{\texttt{STTT}, \texttt{SFTT}, \texttt{STTF}, \texttt{SFTF}\}$$

$$T_{N1} = \{\texttt{STFT}, \texttt{STFF}, \texttt{SFFF}\}$$

$$T_{P2} = \{\texttt{STFT}\}$$

$$T_P = \{\texttt{STTT}, \texttt{SFTT}, \texttt{STTF}, \texttt{SFTF}, \texttt{STFT}\}$$

$$C_3 = (\neg x_1 \vee \neg x_3 \vee x_4)$$

x_4 matches v_3

$$T_{P1} = \{\texttt{SFTT}, \texttt{SFTF}, \texttt{SFFF}\}$$

$$T_{N1} = \{\texttt{STTT}, \texttt{STFT}, \texttt{STTF}, \texttt{STFF}\}$$

$$T_{P2} = \{\texttt{STTF}, \texttt{STFF}\}$$

$$T_N = \{\texttt{SFTT}, \texttt{SFTF}, \texttt{SFFF}, \texttt{STTF}, \texttt{STFF}\}$$

Append

$$T_P = \{\texttt{STTT}, \texttt{STFT}, \texttt{SFTT}\}$$

$$T_N = \{\texttt{STTF}, \texttt{STFF}, \texttt{SFTF}, \texttt{SFFF}\}$$

Mix

$$T = \{\texttt{STTT}, \texttt{STFT}, \texttt{SFTT}, \texttt{STTF}, \texttt{STFF}, \texttt{SFTF}, \texttt{SFFF}\}$$

B.4 Distribution Algorithm

Initialize the tube T with the variables from the first clause

$$T = \{(x_1), (x_2), (\neg x_3)\}$$

Select Clause 2:

$T_1 = \textsc{Insert Literal}(T, x_2)$

$$T_1 = \{(x_1, x_2), (x_2), (x_2, \neg x_3)\}$$

$T_2 = \textsc{Insert Literal}(T, x_3)$

$$T_2 = \{(x_1, x_3), (x_2, x_3)\}$$

$T_3 = \textsc{Insert Literal}(T, \neg x_4)$

$$T_3 = \{(x_1, \neg x_4), (x_2, \neg x_4), (\neg x_3, \neg x_4)\}$$

$T = \text{mix}(T_1, T_2, T_3)$

$$T = \{(x_1, x_2), (x_2), (x_2, \neg x_3), (x_1, x_3), (x_2, x_3), (x_1, \neg x_4), (x_2, \neg x_4), (\neg x_3, \neg x_4)\}$$

Select Clause 3:

$T_1 = \textsc{Insert Literal}(T, \neg x_1)$

$$T_1 = \{(\neg x_1, x_2), (\neg x_1, x_2, \neg x_3), (\neg x_1, x_2, x_3), (\neg x_1, x_2, \neg x_4), (\neg x_1, \neg x_3, \neg x_4)\}$$

$T_2 = \textsc{Insert Literal}(T, \neg x_3)$

$$T_2 = \{(x_1, x_2, \neg x_3), (x_2, \neg x_3), (x_2, \neg x_3), (x_1, \neg x_3, \neg x_4), (x_2, \neg x_3, \neg x_4), (\neg x_3, \neg x_4)\}$$

$T_2 = \textsc{Insert Literal}(T, x_4)$

$$T_3 = \{(x_1, x_2, x_4), (x_2, x_4), (x_2, \neg x_3, x_4), (x_1, x_3, x_4), (x_2, x_3, x_4)\}$$

$T = \text{mix}(T_1, T_2, T_3)$

$$\begin{aligned} T = \{&(\neg x_1, x_2), (\neg x_1, x_2, \neg x_3), (\neg x_1, x_2, x_3), (\neg x_1, x_2, \neg x_4), (\neg x_1, \neg x_3, \neg x_4), \\ &(x_1, x_2, \neg x_3), (x_2, \neg x_3), (x_1, \neg x_3, \neg x_4), (x_2, \neg x_3, \neg x_4), (\neg x_3, \neg x_4), \\ &(x_1, x_2, x_4), (x_2, x_4), (x_2, \neg x_3, x_4), (x_1, x_3, x_4), (x_2, x_3, x_4)\} \end{aligned}$$