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 5, 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 witness candidates in exponential time.

Molecular computation requires exponential space inorder 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. 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 potentially satisfying witness for a SATISFIABILITY instance is a Boolean assignment to the variables that makes the formula true. Such a witness can be represented as an n-bit Boolean vector. 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 $\phi$
  do {
    $test\_clause \leftarrow False$
    for each literal $\ell$ in $C$
      do {
        if $\ell \in B$
          then $test\_clause \leftarrow True$
      }
    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]. The function COMBINATORIALGENERATE(n), which we introduce in Chapter 3, constructs an exponential number of witness candidates in linear time.

Algorithm 1.1.2: BRUTESAT(ϕ)

```
$n$ number of variables in $\phi$
$t$ is bit-vector representing a witness candidate

for $t \leftarrow 0$ to $2^n - 1$
  do if CHECKSAT($\phi, t$)
       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 test configuration t satisfies the input instance ϕ, then the algorithm returns `SATISFIABLE`; otherwise the algorithm returns `UNSATISFIABLE`.

Algorithm 1.1.3: EXTRACTSAT(ϕ)

```
$n$ number of variables in $\phi$

$T \leftarrow$ COMBINATORIALGENERATE($n$)
for each clause $C$ in $\phi$
  do $T_C \leftarrow \emptyset$
     for each literal $\ell$ in $C$
       do $T_T \leftarrow$ extract($T, \ell$)
          $T_C \leftarrow$ mix($T_C, T_T$)
     $T \leftarrow T_C$
if $T = \emptyset$
  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). 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 algorithm [16], Ogihara and Ray's algorithm [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 manufacturing 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 On microbiology and computation

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.

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`$5'$

Suppose 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. The resulting oligonucleotide O is defined

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

Molecular computation uses oligonucleotides for both storing and operating on a problem state. These operations include matching and replication. Although this setting 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'ATCTTT3'
    3'AAAGGA5'.
```

Next we attach v_2 to the resulting complex, yielding

```
5'ATCTTT|CCTATA3'
    3'AAAGGA5'.
```

Finally the edge may be removed and we have the sequence

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

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*
$T \leftarrow mix(T_1, \ldots, T_n)$ — *combine n test tubes of information. 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*
$T' \leftarrow append(T, s)$ — *the concatenation of the oligonucleotide s with each element in T.*

Definition 2.3.4. *Split*
$[T', T''] \leftarrow split(T)$ — *portions T into two tubes. 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*
$T' \leftarrow extract(T, s)$ — *separates all oligonucleotides from T containing the sequence 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. 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*
$detect(T)$ — *determine if any encodings are present in 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*
$[a_1, a_2] \leftarrow$ *splice*(a, b) — *cuts an oligonucleotide* $a = a_1 \cdot b \cdot a_2$ *with a subsequence* b *into two pieces by a restriction enzyme. 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*
$T' \leftarrow$ *purify*(T) — *provides a uniform distribution from the contents of* T *as* 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 independently introduced 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 polynomial time reduced from any NP language [23]. An NP-hard problem instance includes the Halting Problem [23], which determines if any program can terminate. Witnesses for Satisfiability can be expressed in a polynomial on the input ϕ. The output for Satisfiability consists of either all witnesses or a single witness if the problem instance is satisfiable; otherwise the problem instance is unsatisfiable.

Validation of a Satisfiability instance can use a witness candidate on a circuit. Let us consider a circuit for Satisfiability with 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.

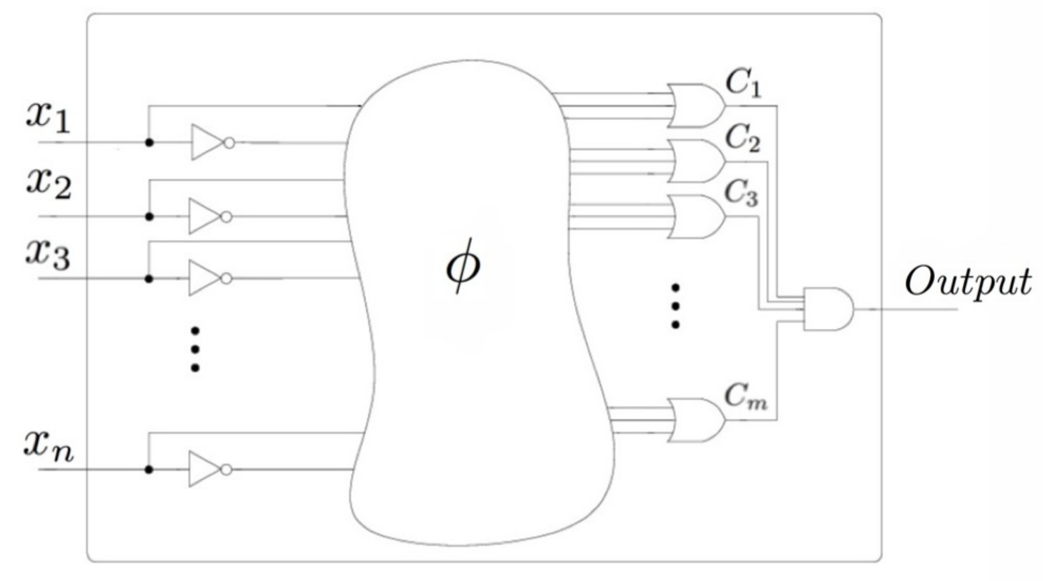

Figure 2.1: A circuit describing SATISFIABILITY.

The realization of SATISFIABILITY as a circuit shows 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. Subproblems of SATISFIABILITY include CNF, k-CNF, and k-SAT problem definitions.

Definition 2.4.2. *CNF*
Conjunctive Normal Form consists of the intersection of sets of disjunctive literals.

Definition 2.4.3. *k-CNF*
Consists of a CNF instance with each disjunctive clause containing k literals.

Definition 2.4.4. k-SAT
Problem variant of SATISFIABILITY *where each clause consists of k Boolean literals. k-CNF formula provide an equivalent representation.*

2.5 Evaluating SAT Solvers

In this section, we describe two standards for encoding the SATISFIABILITY problem that we adopt. This includes the input and output 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.

Definition 2.5.1. SAT *phase transition*
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 transition is a region where both satisfiable and unsatisfiable instances are likely [12]. This region separates trivially satisfiable and over-constrained unsatisfiable SATISFIABLE problem instances. SATISFIABILITY instances with low α are under-constrained and trivially satisfiable; those instances with high α are over-constrained and trivially unsatisfiable [12].

Definition 2.5.2. Sat *backbones*
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 of Satisfiability instances [22]: random assignment, combinatorial problems, and industrial applications.

Random Sat

A random k-Sat instance consists of m clauses with k literals per-clause from n variables [25]. An instance is drawn uniformly over the set of variables without replacement.

Hard combinatorial Sat

Combinatorial problem instances provide 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 in an efficient amount of time.

Chapter 3

Existing molecular algorithms for Satisfiability

In this chapter, we introduce two molecular algorithms for Satisfiability. These algorithms construct a set of all witnesses for Satisfiability instances. Lipton's algorithm requires a combinatorial space of all witness candidates to be constructed before execution. Ogihara and Ray's algorithm constructs a set of witness candidates during execution. Following the description, we explore the physical implementations of—and simulation frameworks for—these algorithms.

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 3.1.2) first constructs a combinatorial space containing oligonucleotide configurations for all potential witness candidates. Evaluation of a Satisfiability instance extracts witnesses for the current clause. Reading each clause constrains the contents of the tube T to only those configurations that satisfy the current clause. The algorithm terminates with a set of witnesses for the CNF input ϕ. If ϕ is unsatisfiable, then $T = \emptyset$.

3.1.1 Description of Lipton's algorithm

The function Combinatorial Generate (Algorithm 3.1.1) implements the split-mix synthesis technique [9, 10]. Combinatorial Generate returns a tube T_{comb} consisting of 2^n oligonucleotides that represent potential witness candidates. The tube T_{comb} begins with an initial medium denoted by `S`. An iterative loop extends T_{comb} using split-mix synthesis.

Each split corresponds with appending the tubes with a true (T) and false (F) assignment. The two tubes are mixed and amplified to contain equimolar portions.

Algorithm 3.1.1: COMBINATORIAL GENERATE(n)

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

Figure 3.1: 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 S initiates the tube T_{comb} with a medium for combinatorial synthesis.
Contents of T_{comb}

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

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

$$\begin{aligned} T_1 &= \{\texttt{ST}, \texttt{SF}\}, \text{ and} \\ T_2 &= \{\texttt{ST}, \texttt{SF}\}. \end{aligned}$$

Next append each of the tubes with a positive (`T`) and negative (`F`) assignment for the literal v_2. We have

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

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.

Algorithm 3.1.2: LIPTON'S ALGORITHM(ϕ)

```
$T \leftarrow$ COMBINATORIAL GENERATE($n$)
for each clause $C$ in $\phi$
    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
                        $T_N \leftarrow \text{extract}(T, v_{\mathtt{F}})$
                        $T_c \leftarrow \text{mix}(T_N, T_c)$
        $T \leftarrow T_c$
return (detect($T$))
```

Figure 3.2: LIPTON'S ALGORITHM iterates over each of the m clauses. The contents of T_C incrementally grows 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 satisfiable instances for 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 parsing a 3-CNF formula. In this section, we describe the preconditions and execution of Ogihara and Ray's algorithm.

3.2.1 Description of Ogihara and Ray's algorithm

Ogihara and Ray's algorithm requires two attributes for CNF input:

1. All clauses consist of exactly three literals
2. All clauses must be sorted by the literal's variable

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

Sorting the clauses fulfills Attribute (2). The polarity of each literal may consist of a positive or negative assignment, providing the weak ordering

$$v_1 < \cdots < v_n,$$

Algorithm 3.2.1: Ogihara and Ray's Algorithm(ϕ)

```
$n$ number of variables in $\phi$
Access each sorted literal $[a, b, c]$ from the clause $C$

$T \leftarrow \{\texttt{STT}, \texttt{STF}, \texttt{SFT}, \texttt{SFF}\}$
for $v \leftarrow 3$ to $n$
  do
    $[T_P, T_N] \leftarrow$ split($T$)
    for each clause $C$ in $\phi$
      do
        $[a, b, c] \leftarrow C$
        if $v_{\texttt{T}} = c$
          then
            $T_{P1} \leftarrow$ extract($T_N, a_{\texttt{T}}$)
            $T_{N1} \leftarrow$ extract($T_N, a_{\texttt{F}}$)
            $T_{P2} \leftarrow$ extract($T_{N1}, b_{\texttt{T}}$)
            $T_N \leftarrow$ mix($T_{P1}, T_{P2}$)
        if $v_{\texttt{F}} = c$
          then
            $T_{P1} \leftarrow$ extract($T_P, a_{\texttt{T}}$)
            $T_{N1} \leftarrow$ extract($T_P, a_{\texttt{F}}$)
            $T_{P2} \leftarrow$ extract($T_{N1}, b_{\texttt{T}}$)
            $T_P \leftarrow$ mix($T_{P1}, T_{P2}$)
    $T_P \leftarrow$ append($T_P, v_{\texttt{T}}$)
    $T_N \leftarrow$ append($T_N, v_{\texttt{F}}$)
    $T \leftarrow$ mix($T_P, T_N$)
return (detect($T$))
```

Figure 3.3: 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.

Expanding each partial assignment iterates over each clause in the input CNF. Construction of witness candidates consider the possibilities of the clause ordering

$$a < b < c.$$

Ogihara and Ray's Algorithm begins with four 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 $x_1 = a_\texttt{T}$

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

From T, Select configurations that satisfy $\neg x_1 = a_\texttt{F}$

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

From T_{N1}, select configurations that satisfy $\neg x_2 = b_\texttt{F}$

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

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

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

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 introduce existing implementations of molecular SATISFIABILITY solvers. Physical implementations apply molecular biology techniques using real molecules. Simulation frameworks 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 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 initially consists of single literal witnesses from a single clause. Literals from each clause get distributed into non-conflicting witness candidates. A witness candidate conflicts when it contains both x_i and $\neg x_i$. In this case the configuration gets removed from the set of non-conflicting witnesses.

4.1.1 Description of the Distribution algorithm

The Distribution Algorithm (Algorithm 4.1.1) starts with the literal assignments of a clause. Evaluation of subsequent clauses extends the solution space using the Insert Literal subroutine (Algorithm 4.1.2). During each insertion, the literal gets inserted into a potential solution vector. Table 4.1 lists the five possibilities for literal assignment.

Algorithm 4.1.1: DISTRIBUTION ALGORITHM(ϕ)

```
$m$ number of clauses
$k$ number of literals in each clause

Initialize with the literals from the first clause
$T \leftarrow \{C_1\}$
for $i \leftarrow 2$ to $m$
    do
        $T_C \leftarrow \emptyset$
        for each literal $\ell$ in $C_i$
            do
                $T_I \leftarrow$ INSERT LITERAL($T, \ell$)
                $T_C \leftarrow \mathrm{mix}(T_C, T_I)$
        $T \leftarrow T_C$
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.

Table 4.1: Configurations for the INSERT LITERAL subroutine

Case	**Return configuration**	**State**
1	$\ell \cdot s$	if ℓ is less than all elements in s
2	$s \cdot \ell$	if ℓ is greater than all elements in s
3	$s_1 \cdot \ell \cdot s_2$	if ℓ is between two elements in s
4	$\emptyset$	if ℓ conflicts with $\neg\ell$ in s
5	s	if ℓ exists in s

During this phase, each literal from a disjunctive clause gets considered, incrementally constructing a partial solution space. Cases (1), (2), and (3) place a literal ℓ into an existing sequence s. Each of these cases represents when the literal ℓ get inserted in a non-decreasing sequence.

Algorithm 4.1.2: INSERT LITERAL(T, ℓ)

```
$T_R \leftarrow \emptyset$
for each string $s$ in $T$
do
    case (1) : $\ell < s$
      then $s' \leftarrow \text{append}(\ell, s)$
    case (2) : $\ell > s$
      then $s' \leftarrow \text{append}(s, \ell)$
    case (3) : $s_1 < \ell < s_2$
      then
          $[s_1, s_2] \leftarrow \text{splice}(s, \ell)$
          $s_1 \leftarrow \text{append}(s_1, \ell)$
          $s' \leftarrow \text{append}(s_1, s_2)$
    case (4) : $\neg\ell \in s$
      then $s' \leftarrow \emptyset$
    case (5) : $\ell \in s$
      then $s' \leftarrow s$
    $T_R \leftarrow \text{mix}(T_R, s')$
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.

A literal conflict occurs when both positive and negative assignments of a literal occur in a sequence s. In this case (4), the sequence s gets removed from the set potential solutions. If the sequence s contains the literal ℓ, case (5), then the existing sequence s gets returned unmodified.

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, literals from the second clause get inserted 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 $[x_1]$ is returned. 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

$$T_C = \{[x_1], [x_1, x_2], [x_1, \neg x_2], [x_1, x_3], [x_2, x_3]\}.$$

The contents of T_C get assigned 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 provide an overview of the software and download location for Molecular Simulation and its documentation. We provide tools for use with Molecular Simulation. This includes `Perl` execution scripts and visualization for output data. We provide examples for Molecular Simulation's input and output. Invocation of Molecular Simulation from the command line provides user configurable options. The next chapter describes the usage of Molecular Simulation with automated execution.

5.1 Overview

Molecular Simulation provides a 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 provide a background and pseudocode for these algorithms.

5.2 Download

Molecular Simulation can be downloaded from:
`https://github.com/dncarley/MolecularSimulation`.

5.3 Requirements

Requirements for Molecular Simulation are specified in this section. This includes the hardware and software requirements for running Molecular Simulation on your system.

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 on your system.

`Perl` must be installed on your system 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 provides detailed datatype, function, and class definitions.

5.5 Tools

This project uses several tools for automating tasks and execution. In this section, we introduce 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

A Sat datapoint visualization for Molecular Simulation's output can be downloaded from:
`https://github.com/dncarley/VisualizeSatDatapoints`
Ben Fry's example in Chapter 4 of *Visualizing Data* [8] provides a framework for importing output from Molecular Simulation. The visualization project directory contains a README for usage.

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`

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’
c
c cnf input is designated with ‘p cnf’
c    followed by number of variables <n>, and clauses <m>
c
p cnf <n> <m>
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
-3 9 14 0
6 -9 -12 0
-2 11 17 0
3 -13 -17 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 -3 -9 11 13 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 footprint 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. We would like to execute the 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 building 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. We provide example configuration options for automating the execution of Molecular Simulation. The example continues with an analysis of runtime metrics for each test instance. The next chapter provides 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 will 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. 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. Now we provide the automated invocation for processing datasets with each of the algorithms.

The provided `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

`-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 output can be analyzed after the automated tests have completed. The output consists of the standard Sat Competition output appended with custom runtime metric logging. We discuss viewing output directly during execution and reading saved output files. 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` provides 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 `*.tsv` file. Subsequent datapoint browsing and the online view use the `*.tsv` file for condensed reading and transmission. In the next chapter, we provide the results of the experimental setup.

Chapter 7

Results

This chapter provides results of the k-SAT execution test from the previous chapter. We consider the results of the test and provide analysis of the algorithm metrics.

7.1 Algorithm metric comparison

This section provides results from the simulation. We provide the analysis for the molecular operations. These include counts of append, extract, mix, purify, splice, and split. Presentation of actual computation time and required memory for the solution representation allow for comparison of algorithms.

Append is an operation that concatenates molecules.

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 amount of appends. This depends on the number of variables and clauses present in the CNF instance.

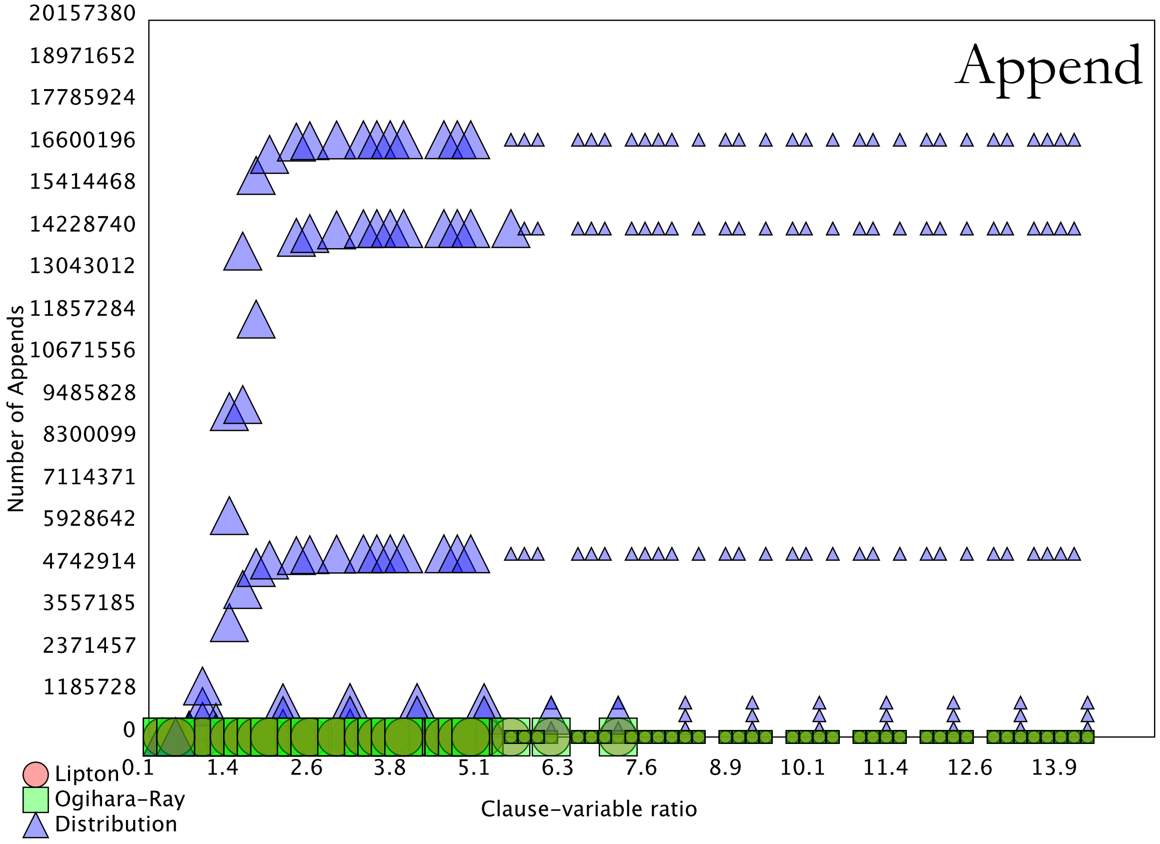

Figure 7.1: Clause to variable ratio α vs. Number of appends

Extract is an operation that filters strings.

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

The Distribution algorithm does not require extract.

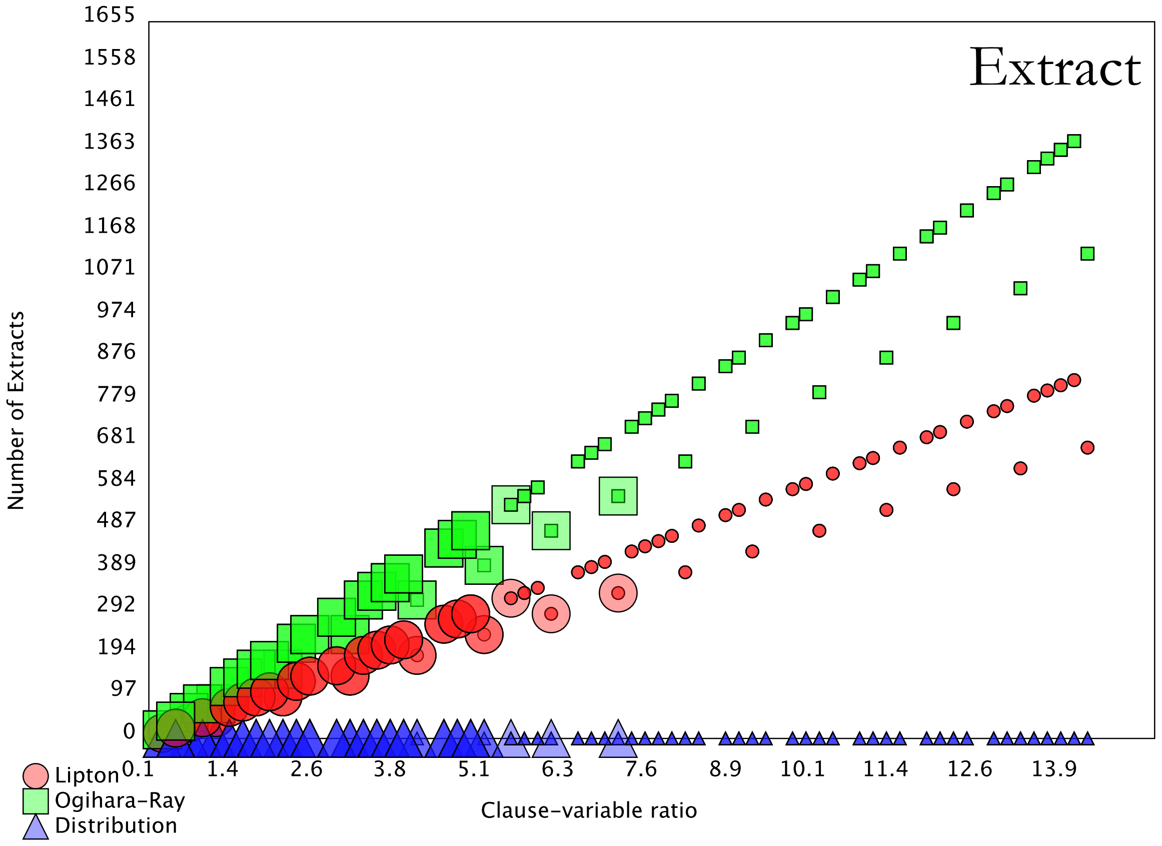

Figure 7.2: Clause to variable ratio α vs. Number of extracts

Mix is an operation that combines two tubes.

Lipton's algorithm requires a linear amount 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 amount of mixes on α.

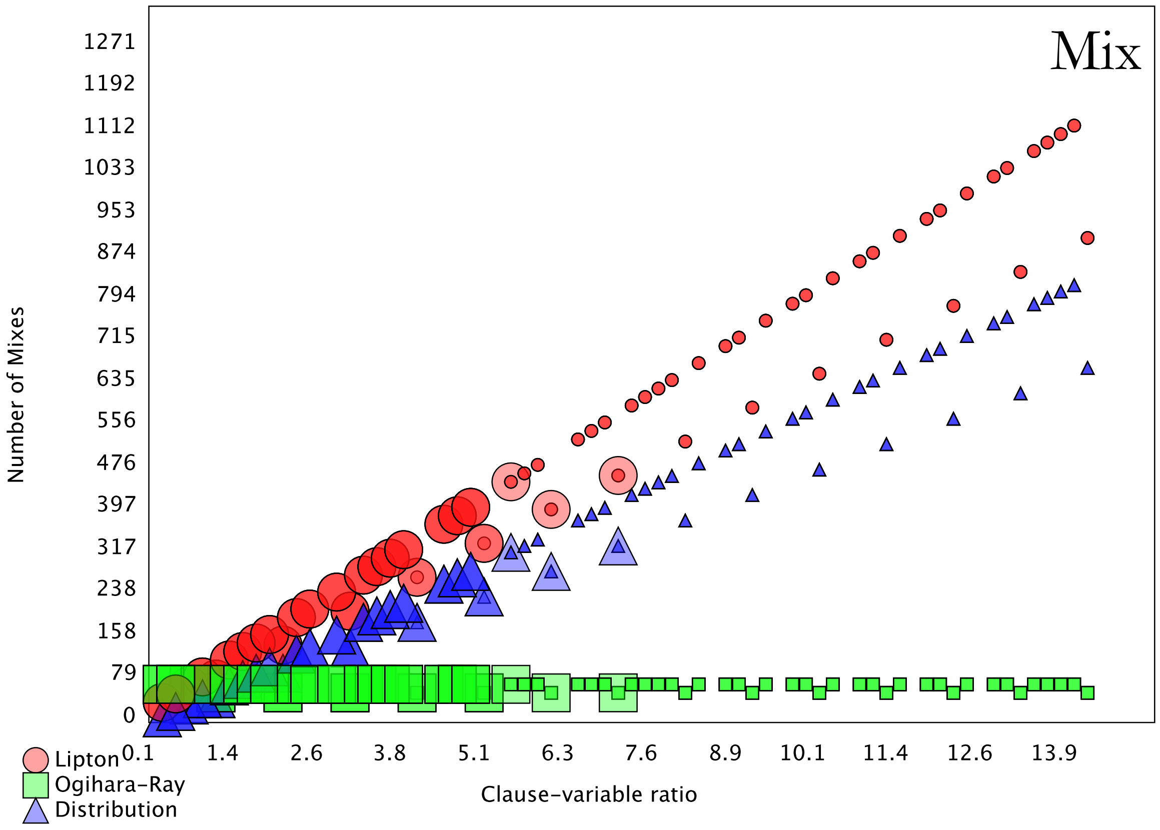

Figure 7.3: Clause to variable ratio α vs. Number of mixes

Purify is an operation that ensures equal portions of each independent string.

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

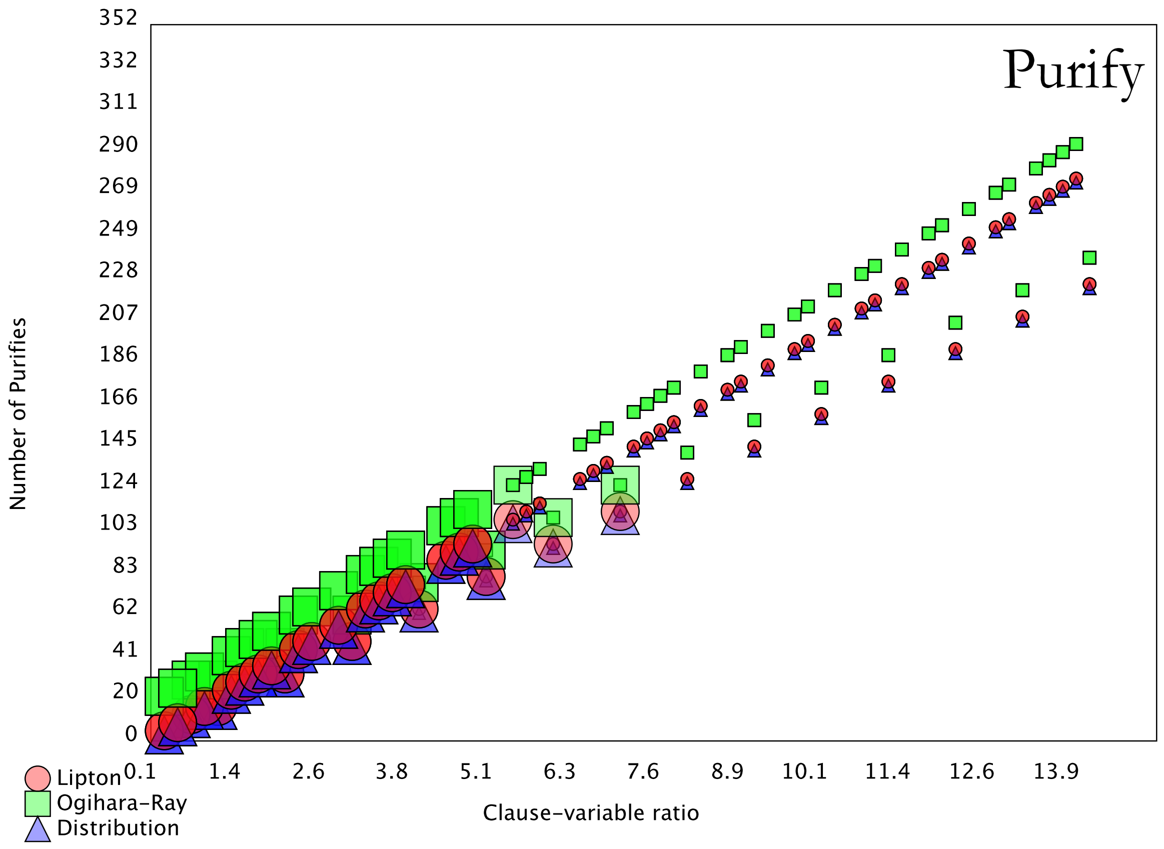

Figure 7.4: Clause to variable ratio α vs. Number of purifies

Splice is an operation that inserts a string 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 7.1 shows the number of appends.

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

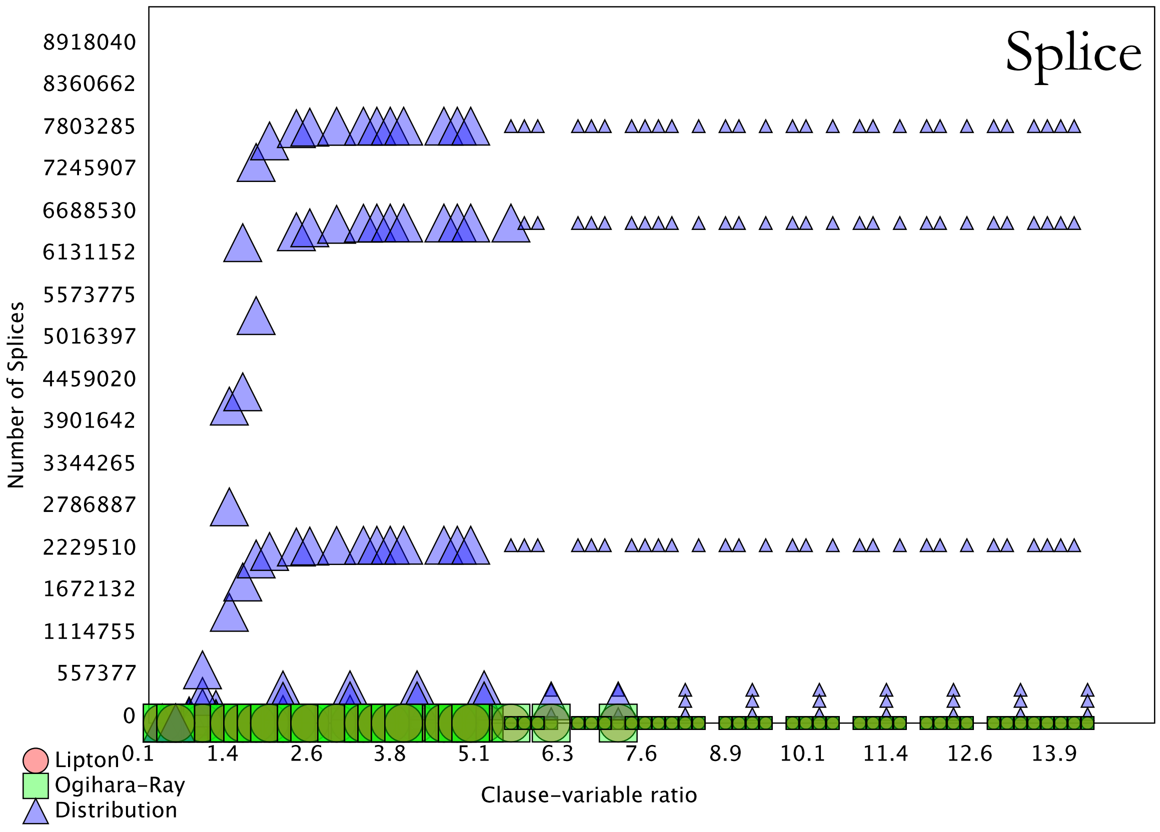

Figure 7.5: Clause to variable ratio α vs. Number of splices

Split is an operation that portions a tube into two exact copes.

Distribution requires a linear number of splits.

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

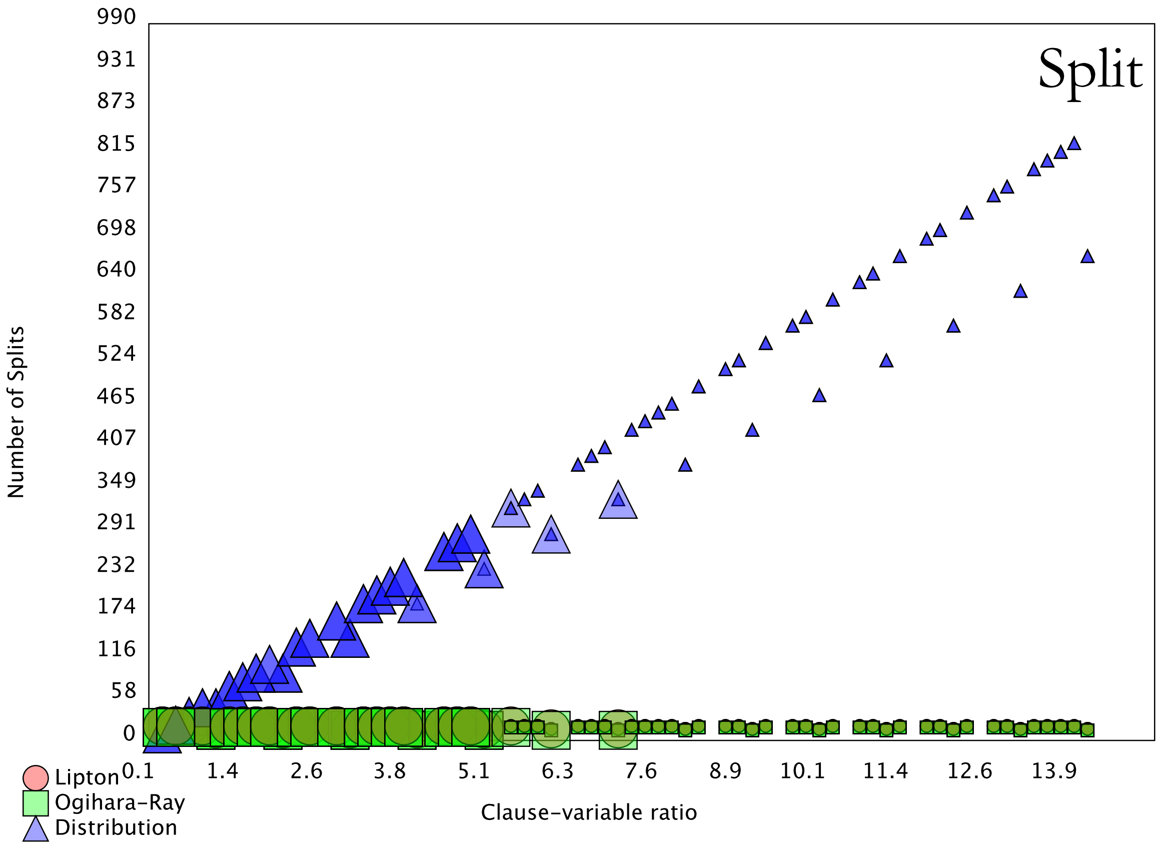

Figure 7.6: Clause to variable ratio α vs. Number of splits

Time is a measurement of algorithm execution in seconds.

Ogihara-Ray's algorithm requires the least amount of time. In cases where the SATISFIABILITY instance is under-constrained, where more possible solutions occur, the algorithm takes the greatest amount of 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.

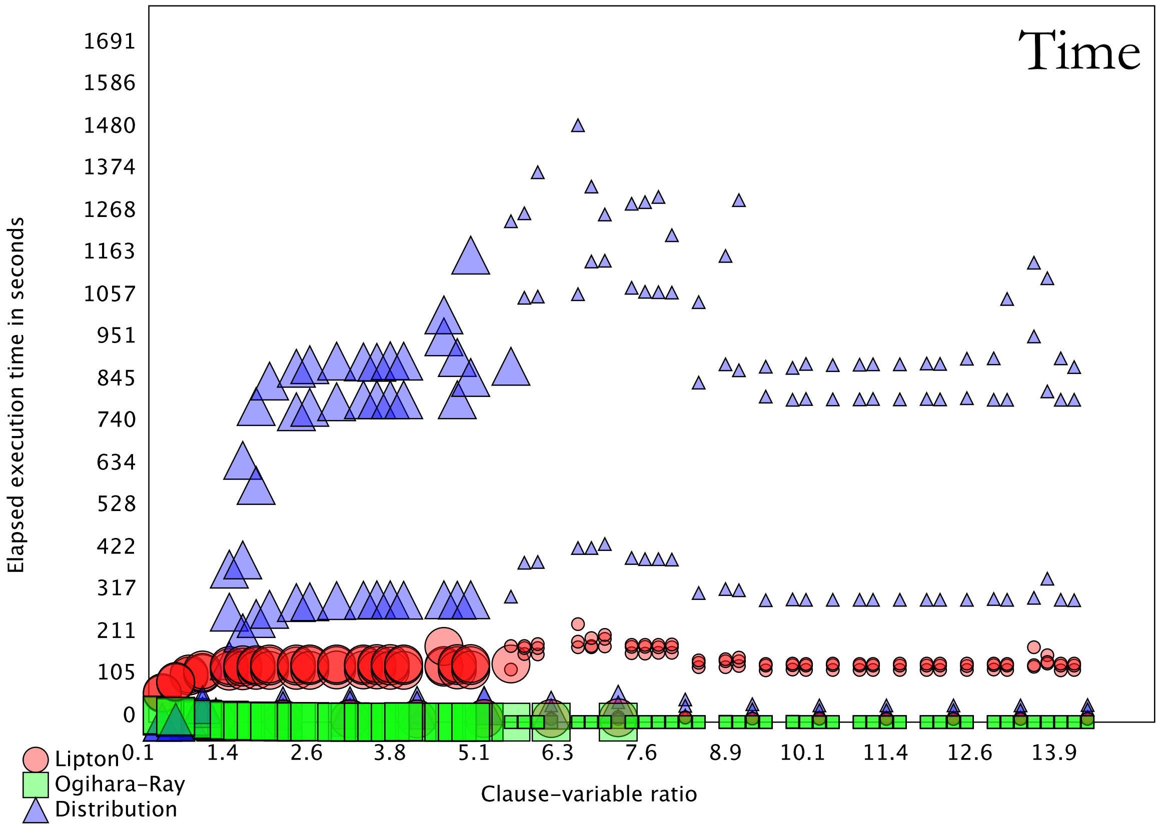

Figure 7.7: Clause to variable ratio α vs. execution time in seconds

Memory is a measurement of the satisfiable instance footprint returned by each algorithm measured 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 provides 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 7.8 scales accordingly.

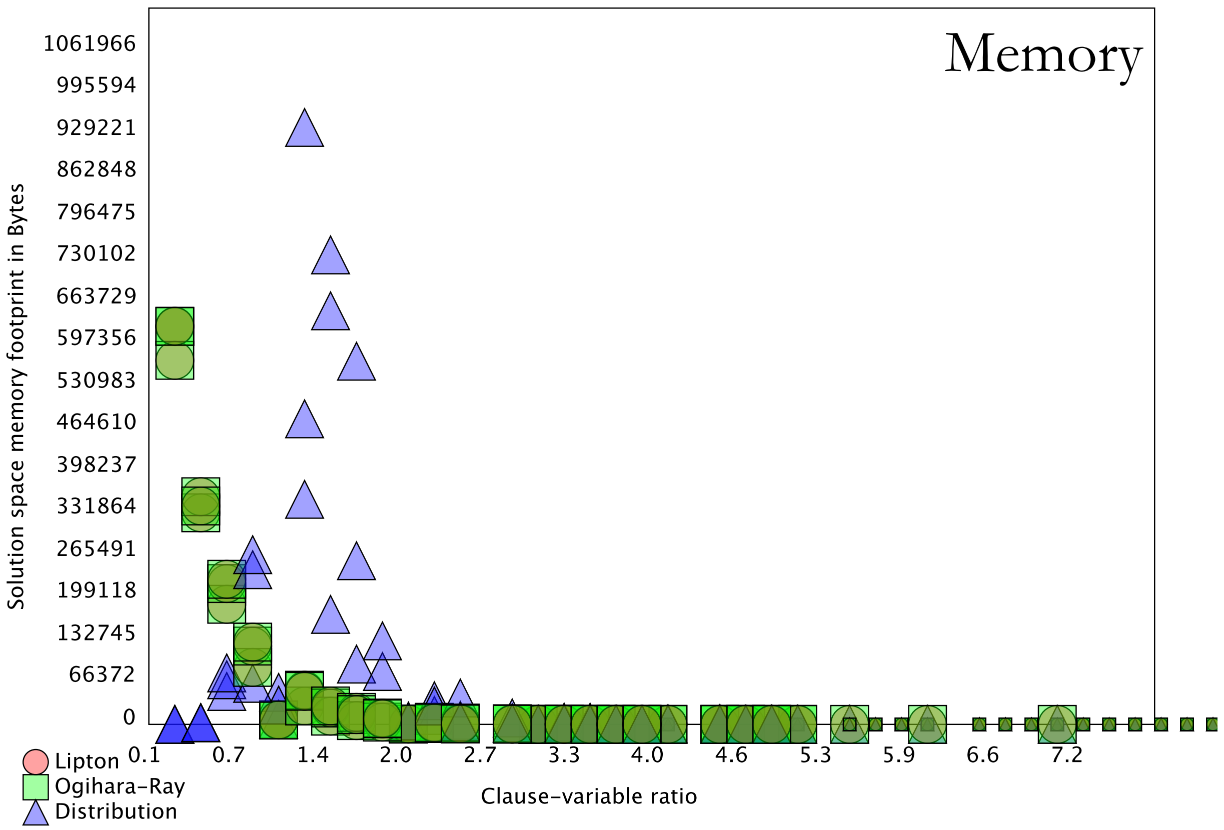

Figure 7.8: Clause to variable ratio α vs. satisfiable solution footprint in Bytes

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 provides 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

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

$$T =$$

```
TTTT FTTT TFTT FFTT TTFT FTFT TFFT FFFT
TTTF FTTF TFTF FFTF TTFF FTFF TFFF FFFF
```

Next select Clause 1:

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

```
TTTT FTTT TFTT FFTT TTFT FTFT TFFT FFFT
TTTF FTTF TFTF FFTF TTFF FTFF TFFF FFFF
```

Extract x_1:

```
TTTT      TFTT      TTFT      TFFT
TTTF      TFTF      TTFF      TFFF
```

Extract x_2:

```
TTTT FTTT           TTFT FTFT
TTTF FTTF           TTFF FTFF
```

Extract $\neg x_3$:

```
                    TTFT FTFT TFFT FFFT
                    TTFF FTFF TFFF FFFF
```

Mix contents:

```
TTTT FTTT TFTT      TTFT FTFT TFFT FFFT
TTTF FTTF TFTF      TTFF FTFF TFFF FFFF
```

Next select Clause 2:

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

```
TTTT FTTT TFTT      TTFT FTFT TFFT FFFT
TTTF FTTF TFTF      TTFF FTFF TFFF FFFF
```

Extract x_2:

```
TTTT FTTT           TTFT FTFT
TTTF FTTF           TTFF FTFF
```

Extract x_3:

```
TTTT FTTT TFTT
TTTF FTTF TFTF
```

Extract $\neg x_4$:

```

TTTF FTTF TFTF      TTFF FTFF TFFF FFFF
```

Mix contents:

```
TTTT FTTT TFTT      TTFT FTFT
TTTF FTTF TFTF      TTFF FTFF TFFF FFFF
```

Finally, select Clause 3:

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

```
TTTT FTTT TFTT      TTFT FTFT
TTTF FTTF TFTF      TTFF FTFF TFFF FFFF
```

Extract $\neg x_1$:

```
     FTTT                FTFT
     FTTF                FTFF      FFFF
```

Extract $\neg x_3$:

```
                         TTFT FTFT
                         TTFF FTFF TFFF FFFF
```

Extract x_4:

```
TTTT FTTT TFTT           TTFT FTFT
```

Mix contents:

```
TTTT FTTT TFTT           TTFT FTFT
     FTTF                TTFF FTFF TFFF FFFF
```

B.3 Ogihara and Ray's Algorithm

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

$$T = \{\texttt{TT}, \texttt{TF}, \texttt{FT}, \texttt{FF}\}$$

Iterate variable x_3:

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

$\neg x_3$ matches v_3

$$\begin{aligned} T_{P1} &= \{\texttt{TT}, \texttt{TF}\} \\ T_{N1} &= \{\texttt{FT}, \texttt{FF}\} \\ T_{P2} &= \{\texttt{FT}\} \\ T_P &= \{\texttt{TT}, \texttt{TF}, \texttt{FT}\} \end{aligned}$$

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

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

$$\begin{aligned} T_P &= \{\texttt{TTT}, \texttt{TFT}, \texttt{FTT}\} \\ T_N &= \{\texttt{TTF}, \texttt{TFF}, \texttt{FTF}, \texttt{FFF}\} \end{aligned}$$

Mix

$$T = \{\texttt{TTT}, \texttt{TFT}, \texttt{FTT}, \texttt{TTF}, \texttt{TFF}, \texttt{FTF}, \texttt{FFF}\}$$

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

$$\begin{aligned} T_{P1} &= \{\texttt{TTT}, \texttt{FTT}, \texttt{TTF}, \texttt{FTF}\} \\ T_{N1} &= \{\texttt{TFT}, \texttt{TFF}, \texttt{FFF}\} \\ T_{P2} &= \{\texttt{TFT}\} \\ T_P &= \{\texttt{TTT}, \texttt{FTT}, \texttt{TTF}, \texttt{FTF}, \texttt{TFT}\} \end{aligned}$$

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

x_4 matches v_3

$$\begin{aligned} T_{P1} &= \{\texttt{FTT}, \texttt{FTF}, \texttt{FFF}\} \\ T_{N1} &= \{\texttt{TTT}, \texttt{TFT}, \texttt{TTF}, \texttt{TFF}\} \\ T_{P2} &= \{\texttt{TTF}, \texttt{TFF}\} \\ T_N &= \{\texttt{FTT}, \texttt{FTF}, \texttt{FFF}, \texttt{TTF}, \texttt{TFF}\} \end{aligned}$$

Append

$$\begin{aligned} T_P &= \{\texttt{TTT}, \texttt{TFT}, \texttt{FTT}\} \\ T_N &= \{\texttt{TTF}, \texttt{TFF}, \texttt{FTF}, \texttt{FFF}\} \end{aligned}$$

Mix

$$T = \{\texttt{TTT}, \texttt{TFT}, \texttt{FTT}, \texttt{TTF}, \texttt{TFF}, \texttt{FTF}, \texttt{FFF}\}$$

B.4 Distribution Algorithm

Initialize the tube T with the variables from the first clause

$$T = \{[1], [2], [-3]\}$$

Select Clause 2

$T_1 = \textsc{InsertVariable}(T, 2)$

$$T_1 = \{[1, 2], [2], [2, -3]\}$$

$T_2 = \textsc{InsertVariable}(T, 3)$

$$T_1 = \{[1, 3], [2, 3]\}$$

$T_3 = \textsc{InsertVariable}(T, -4)$

$$T_3 = \{[1, -4], [2, -4], [-3, -4]\}$$

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

$$T = \{[1, 2], [2], [2, -3], [1, 3], [2, 3], [1, -4], [2, -4], [-3, -4]\}$$

Select Clause 3

$T_1 = \textsc{InsertVariable}(T, -1)$

$$T_1 = \{[-1, 2], [-1, 2, -3], [-1, 2, 3], [-1, 2, -4], [-1, -3, -4]\}$$

$T_2 = \textsc{InsertVariable}(T, -3)$

$$T_2 = \{[1, 2, -3], [2, -3], [2, -3], [1, -3, -4], [2, -3, -4], [-3, -4]\}$$

$T_2 = \textsc{InsertVariable}(T, 4)$

$$T_3 = \{[1, 2, 4], [2, 4], [2, -3, 4], [1, 3, 4], [2, 3, 4]\}$$

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

$$\begin{aligned} T = \{&[-1, 2], [-1, 2, -3], [-1, 2, 3], [-1, 2, -4], [-1, -3, -4], \\ &[1, 2, -3], [2, -3], [1, -3, -4], [2, -3, -4], [-3, -4], \\ &[1, 2, 4], [2, 4], [2, -3, 4], [1, 3, 4], [2, 3, 4]\} \end{aligned}$$