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: April 19, 2012

Abstract

Molecular computation is a branch of computing that applies techniques from molecular biology and combinatorial chemistry to perform generalized computations. We explore via simulation in a conventional computer environment molecular algorithms to solve Satisfiability. The simulation environment measures molecular operators for each of the molecular algorithms under test. The test input consists of a set of random 3-Sat instances. The results provide design considerations for an integrated molecular computing architecture.

Contents

Chapter 1

Introduction

Molecular computing uses parallel interactions between genetic molecules, such as DNA or RNA, to perform computational tasks. We provide an experimental environment for simulating three molecular algorithms. This environment simulates the execution of random 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 chapter introduces the contents of the project.

1.1 Introduction to molecular computation

Molecular interactions test many potential states for discrete states of matter. We consider genetic encodings as a witnessing mechanism for computational configurations. Hydrogen bonds form complementary base pairs in DNA and RNA. Complementary genetic string representations encode data for both storage and matching mechanism. Molecular computing takes advantage of molecular interactions for general purpose computation.

We consider three molecular algorithms for solving SATISFIABILITY: Lipton's algorithm [17], Ogihara and Ray's algorithm [20, 21] and a new algorithm, introduced here, that we call the 'Distribution' algorithm. Lipton's algorithm enumerates a combinatorial space and gets reduced to satisfiable solutions. Ogihara and Ray's algorithm constructs a space of potential solutions and eliminates non-satisfiable paths. The Distribution algorithm constructs a set of non-conflicting states for a satisfiable solution. Chapters 3 and 4 discuss the implementation of these algorithms.

1.2 Simulation environment and physical devices

This project introduces a system for simulating three molecular algorithms for solving SATISFIABILITY. The system provides standard operations for molecular computing that we introduce in Chapter 2. The system records runtime metrics, including counts of molecular

operators, solution memory footprint and execution time. These metrics lets us analyze algorithm performance for each SATISFIABILITY test instance.

Existing fabrication techniques for constructing nanopores [16, 22] and micropumps [15] provide the technology for sequencing genes. These gene sequencing technologies, discussed in Chapter 7, provide design considerations for an integrated molecular computing architecture. In the next chapter, we introduce techniques from nanotechnology, microbiology and theoretical computer science for applied 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 encoding information with molecular matter. Following this example, we introduce Adleman's molecular operators for solving an instance of Hamiltonian Path. The operators provide a base instruction set for molecular computing, and provide the primitives to construct molecular algorithms.

Finally, 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 for the test cases that we consider.

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' [8]. Examples of applied nanotechnology include the manufacturing of graphene [26] and DNA nanopores [18]. Graphene consists of an arrangement of carbon atoms that provides desirable physical and electrical properties [26]. DNA nanopores create a physical channel for threading DNA for read and write operations. Diverse applications continue to take advantage of the properties of nanotechnology. Gene sequencing technologies provide an example of applied nanotechnology [16, 22].

Smaller and cost-effective DNA sequencers provide the ability to read the contents of a gene. Benchtop sequencers [16, 22] allows 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 [16, 22].

2.2 On microbiology and computation

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

Strings of nucleotides encode information as oligonucleotides. A *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 the DNA sequence denotes the direction of genetic information. +DNA gets denoted by $5'$—$3'$ and −DNA gets denoted by $3'$—$5'$. We focus on +DNA and −DNA as a substrate for encoding configurations for computational states.

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

Let us consider two techniques for representing information with oligonucleotides. These allow us to encode integer mappings as a fixed width integer sequence.

Representing an integer sequence requires a systematic mapping of an oligonucleotide entry with an integer counterpart. A fixed width representation map independent sequences on a readable boundary. Now we explore an example for encoding an integer sequence with a sequence of oligonucleotides. A sample mapping is provided in Table 2.1.

Table 2.1: A mapping of the integers $[0, 5]$ 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$5'$
5	$5'$CCTTCC$3'$	$3'$GGAAGG$5'$

Suppose that we would like to encode the sequence of integers S as an equivalent oligonucleotide representation O_1.

We have, e.g.,

$$S = [1, 3, 4, 3, 2, 0]$$

and

$$O_1 = 5'\text{AAACCC} \mid \text{CCCTCC} \mid \text{CTTTTC} \mid \text{CCCTCC} \mid \text{GGTAAA} \mid \text{TCTCCC}3'.$$

Recovering the sequence S from O_1 can be done several ways. Because the definition of the sequence exists, we may use the reverse complement to match sequences. Another method

splits the sequence O_1 on the encoding width. In this case, the encoding width is six base pairs. Gene sequencing tools permit reading the sequence and interpretation of the data with Table 2.1.

Molecular computing encodes genetic information for both storage and operations on a problem state. These interactions include matching and replication. Although this setting describes and artificial construction for a machine, the natural encodings of organisms also share the mechanics that we exploit. Interactions of molecules provide mechanics for generalized computation with oligonucleotides.

In the following chapters, we describe molecular algorithms for SATISFIABILITY and provide insight to construction of a generalized molecular computer. Next we provide a toolbox for molecular computation. The tools presented permit generalized computation with molecular biology techniques. In the next section we introduce the techniques from Adleman's molecular toolbox [1].

2.3 Adleman's molecular toolbox for solving HAMITONIAN PATH

Leonard Adleman performed the first molecular computation in 1994 with 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 HAMILTONIAN PATH
Given an undirected graph G, does there exist a path that visits every vertex exactly once?

Adleman's encoding for graphs uses oligonucleotides for defining each vertex. The vertex representation shares a similar definition for our example of encoding a sequence of integers in Table 2.1. Representing edges requires a definition of a reverse-complement oligonucleotide; this string connects the suffix of the vertex v_i with the prefix of v_j. For example, let us consider an example for appending v_2 to v_1. Let, e.g.,

$$v_1 = 5'\text{ATCTTT}3'$$
$$v_2 = 5'\text{CCTATA}3'.$$

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

$$e_{1,2} = 3'\text{AAATTC}5'.$$

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

$$5'\text{ATCTTT}3'$$
$$\quad\quad 3'\text{AAATTC}5'.$$

Next we attach v_2 to the resulting complex, yielding

$$5'\text{ATCTTT}|\text{CCTATA}3'$$
$$3'\text{AAATTC}5'.$$

Finally the edge may be removed and we have the sequence

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

The sequence $v_1 \cdot v_2$ represents the path v_1 to v_2, and can be obtained with the *append* operation. Possible solutions get stored in a test tube T. T begins as an empty tube. For solving HAMITONIAN PATH, we introduce equimolar portions of each oligonucleotide vertex for a starting configuration with the *mix* operation.

Definition *Mix*
$T \leftarrow \text{mix}(T_1, T_2)$ — combines two test tubes of information. The output consists of a single set $T = T_1 \cup T_2$.

A small initial set may be amplified with *polymerase chain reaction* (PCR). PCR thermocycles the contents of the tube to replicate the contents. Possible paths get randomly generated by introducing the vertex representation to the contents. We create this representation elongating the initial vertex with a fixed path length.

Append attaches a string to each string contained in a test tube. *Split* portions a tube into multiple portions. We will use split-mix synthesis as a technique for generation of combinatorial space in Chapter 3.

Definition *Append*
$T' \leftarrow \text{append}(T, a)$ — the concatenation of the oligonucleotide a with each element in T.

Definition *Split*
$[T', T''] \leftarrow \text{split}(T)$ — distributes T into two tubes. Each of the resulting tubes, T' and T'', contain the same representative elements of T.

From the tube T, we keep paths that begin with V_{in} and end with V_{out}. This ensures that the initial and terminal conditions for the graph get satisified. Extracting only strings from T that match these conditions reduces the number of potential strings.

Definition *Extract*
$T' \leftarrow \text{extract}(T, a, x)$ — separates all oligonucleotides from T containing the sequence a at position x. The output consists of a set T' of those oligonucleotides containing a at position x.

The tube T consists of possible encodings that have the correct starting and ending vertices. We select only strings with length n, where n is the number of vertices in G, to ensure that all vertices get traversed. This can be performed with *gel electrophoresis*, a technique for sorting molecules by mass. Next, we ensure that each vertex occurs exactly once. This gets accomplished by extracting possible vertices. If a vertex occurs multiple times in a path, then the string representation gets discarded.

Finally, we check T with *detect* to determine if any valid paths remain. If valid paths exist, then each string may be read for the path assignment.

Definition *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 construction of molecular Satisfiability solvers. The Distribution algorithm, introduced in Chapter 4, requires the *splice* operation.

Definition *Splice*
$[a_1, a_2] \leftarrow$ splice(a, b) — cuts an oligonucleotide a 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 environment, 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 *Purify*
$T' \leftarrow$ purify(T) — provides a uniform distribution from the contents of T as T'.

2.4 Definition of Satisfiability

Satisfiability is a canonical NP-complete language. Each Satisfiability instance efficiently encodes a set of conditions to satisfy. Because Satisfiability is NP-complete, it can be reduced to any NP-complete language.

Definition Satisfiability
Formally defined as the language

$$\text{Satisfiability} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}[25].$$

Evaluation of a SATISFIABILITY instance requires validating the input with the instance definition. We introduce SATISFIABILITY evaluation with a circuit. Let us consider a three-layered circuit for SATISFIABILITY. 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 expression ϕ. Figure 2.1 contains a schematic for SATISFIABILITY.

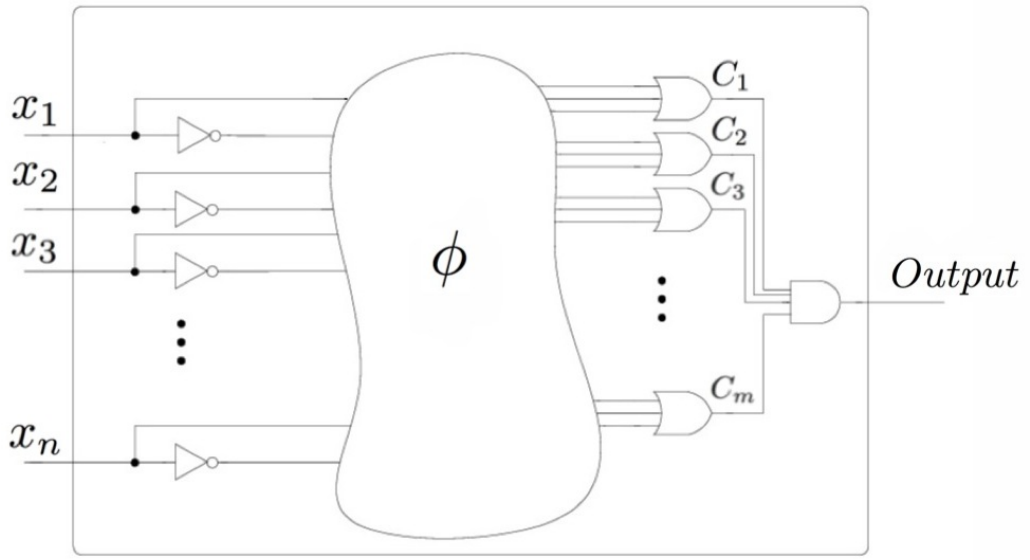

Figure 2.1: A circuit describing SATISFIABILITY.

The realization of SATISFIABILITY as a circuit shows two insights of the problem. SATISFIABILITY can be implemented with logic components proportional to the problem size, and the worst case verification consists of enumerating all possible switch configurations. SATISFIABILITY as a language demonstrates that it is equivalent to all other NP-complete languages.

Cook and Levin independently introduced the canonical instance of a NP-complete language SATISFIABILITY[4, 14]. A NP-complete language is one that is in NP and NP-hard. A NP-hard language is at least as hard as any problem in NP.

The next section considers standards adopted for SATISFIABILITY. This allows practitioners to apply SATISFIABILITY in various settings.

2.5 Evaluating SATISFIABILITY

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

SATISFIABILITY Competition [5, 24].

Next, we introduce problem instance classification for SATISFIABILITY. Classification of SATISFIABILITY problem instances include randomly generated, combinatorial, and industrial [24]. The experimental setup in Chapter 6 considers generation of random k-SAT input.

2.5.1 Input and output

Each year a competition showcases techniques for evaluating SATISFIABILITY [24]. We conform to the standards of the SAT Competition `http://www.satcompetition.org/`. SAT solvers demonstrate state-of-the-art techniques for solving three main tracks of SATISFIABILITY instances. The tracks exhibit applications for SATISFIABILITY, including: industrial applications, hard combinatorial, and random problem instances.

The input and output standards for SATISFIABILITY allow common benchmarks for SAT solvers.

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). The format is user readable with a natural encoding for SATISFIABILITY. We provide an example of this encoding in Section 5.5.

Output

SAT Competition output consists of the status for a DIMACS CNF input instance [24]. This includes the known state, either `SATISFIABLE`, `UNSATISFIABLE`, or `UNKNOWN`. When a witnessing satisfying assignment occurs, the assignment gets provided as a list of integers with the `SATISFIABLE` state. We provide an example along with a custom interface in Section 5.6.

2.5.2 Metrics for classifying SATISFIABILITY

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

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

Definition k-CNF
Consists of a CNF expression with each disjunctive clause containing k literals.

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

Definition SAT phase transition
The SAT phase transition is a region where both satisfiable and unsatisfiable instances are likely. The ratio of clauses to variables $\alpha = m/n$ provides a characterization for where phase transitions may occur in the space of all k-CNF formula [6, 12].

Definition SAT backbones
SAT backbones are the variable assignments present in all of the satisfying assignments to a SATISFIABILITY expression [29]. This is a set of variables that occur in all satisfiable valuations for an input expression.

2.5.3 SATISFIABILITY instances

There are several methods for constructing SATISFIABILITY instances. We consider techniques for constructing instances based on random assignment, combinatorial, and real applications from industry. The instance type demonstrate properties of SATISFIABILITY and provide heuristics for certain input.

A random k-SAT expression consists of m clauses with k literals per-clause from n variables [27]. Variable assignments get distributed with probability $Prob\left(\frac{1}{n}\right)$. The positive or negative variable polarity get assigned with a probability $Prob\left(\frac{1}{2}\right)$.

Combinatorial instances provide difficult benchmark cases. These instances can be converted from other NP-complete problems. This category also includes games and graph theoretic problems represented as SATISFIABILITY.

Industrial processes apply SATISFIABILITY in many real world problems. This includes circuit layout, planning, logistics, circuit fault testing and many other industrial NP-complete problems. Applications for industrial SAT will 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 are distinct in the resolution of a SATISFIABILITY instance. Lipton's algorithm requires a space to be constructed before execution, where Ogihara and Ray's algorithm constructs a valid space during execution. Following the description, we explore the physical implementation and simulation of these algorithms.

3.1 Lipton's algorithm for SATISFIABILITY

Introduced in 1995 by Richard Lipton [17], this algorithm creates an exponential search space for the CNF expression. Each variable gets evaluated with the combinatorial space, reducing the space on each iteration. The satisfiable configurations are present in the remaining space. This algorithm is analogous to a conventional brute-force search for all solutions.

3.1.1 Description of Lipton's algorithm

Lipton's algorithm consists of two main procedures. The first phase constructs a combinatorial space of 2^n independent vectors. Second, the combinatorial space gets reduced based on the input CNF instance.

The function COMBINATORIAL GENERATE(n) implements the split-mix synthesis technique [10, 11]. It returns a gel consisting of 2^n independent oligos that correspond to a unique vector space. The space begins construction with an initial medium. An iterative loop elongates a growing solution with the split-mix synthesis. Each split corresponds with appending the tubes with a truth and false assignment. The two tubes are mixed and amplified to contain equimolar portions.

The amplification process gets modeled with a purification step. This eliminates all redundant strings for the simulated environment. After the iteration completes, the complete combinatorial space gets returned. This space consists of 2^n vectors of length n.

From the combinatorial space, we will begin to filter satisfying solutions to the input CNF formula. For each clause, we extract each of the variables present in the solution space. A disjunctive set T_C contains the satisfied string instances for each clause. Lipton's Algorithm iterates over each of the clauses. From the selected clause, the variables get extracted from the combinatorial space. Once complete, the remaining space, T, contains satisfiable instances for ϕ.

3.1.2 Pseudocode for Lipton's algorithm

Algorithms 3.1.1 and 3.1.2 provide pseudocode for Lipton's algorithm.

Algorithm 3.1.1: Combinatorial Generate(n)

$T_{comb} \leftarrow \emptyset$
$T_{comb} \leftarrow \text{mix}(T_{comb}, start)$
for $i \leftarrow 1$ to n
do $\begin{cases} [T_1, T_2] \leftarrow \text{split}(T_{comb}) \\ T_1 \leftarrow \text{append}(T_1, POS) \\ T_2 \leftarrow \text{append}(T_2, NEG) \\ T_{comb} \leftarrow \text{mix}(T_1, T_2) \end{cases}$
return (T_{comb})

Algorithm 3.1.2: Lipton's Algorithm(ϕ)

$T \leftarrow$ Combinatorial Generate($|\ \phi\ |$)
for each clause C in ϕ
do $\begin{cases} T_c \leftarrow \emptyset \\ \textbf{for each} \text{ variable } v \text{ in } C \\ \quad \textbf{do} \begin{cases} \textbf{if } v \text{ is a positive literal} \\ \quad \textbf{then} \begin{cases} T_P \leftarrow \text{extract}(T, POS, v) \\ T_c \leftarrow \text{mix}(T_P, T_c) \end{cases} \\ \quad \textbf{else} \begin{cases} T_N \leftarrow \text{extract}(T, NEG, v) \\ T_c \leftarrow \text{mix}(T_N, T_c) \end{cases} \end{cases} \\ T \leftarrow T_c \end{cases}$
return (detect(T))

3.2 Ogihara and Ray's algorithm for SATISFIABILITY

Ogihara and Ray's algorithm consist of a breadth-first evaluation of clauses from a CNF formula [20, 21]. The algorithm constructs a set of potential solutions 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

Prior to execution of the algorithm it requires two attributes of CNF input:

1. All clauses consist of exactly three literals
2. All clauses must be sorted by variable

Considering only 3-SAT expressions ensure attribute (1) gets fufilled. If models of k-SAT with $k > 3$, then a reduction to 3-SAT must occur prior to execution.

Prior to execution of the algorithm, the parsing of the DIMACS CNF input gets sorted. This step ensures that attribute (2) gets satisfied. Providing the weak ordering

$$v_1 < \cdots < v_n,$$

where the polarity of each variable may consist of a positive or negative valuation.

The initial tube consists of potential states for the first two variables.

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

$$x_u < x_v < x_w.$$

OGIHARA AND RAY'S ALGORITHM evaluates each subsequent variable and determines possible assignments. The possible assignments for the variables v_1 and v_2 get extracted if v_3 matches. Effectively pruning only potential solutions. These potential solutions T_P and T_N get appended with the positive (POS) or negative (NEG) string assignments. The algorithm continues until each variable gets evaluated.

The remaining space T, once the algorithm terminates, contains all solutions for the CNF instance ϕ.

3.2.2 Pseudocode for Ogihara and Ray's algorithm

Algorithm 3.2.1 provides pseudocode for Ogihara and Ray's algorithm.

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

m number of clauses
n number of variables
Each variable of the reordered clause can be accessed by v_1, v_2, and v_3

Reorder variables by most frequent to least frequent literal appearance
Reorder each clause in increasing literal order

$T \leftarrow \{POS \cdot POS, POS \cdot NEG, NEG \cdot POS, NEG \cdot NEG\}$
for each variable x_i in $3 \leq i \leq n$
do
- $[T_P, T_N] \leftarrow \text{split}(T)$
- **for each** clause C in ϕ
  - **do**
    - $[v_1, v_2, v_3] \leftarrow C$
    - **if** $x_i = v_3$
      - **then**
        - $T_{P1} \leftarrow \text{extract}(T_N, POS, v_1)$
        - $T_{N1} \leftarrow \text{extract}(T_N, NEG, v_1)$
        - $T_{P2} \leftarrow \text{extract}(T_{N1}, POS, v_2)$
        - $T_N \leftarrow \text{mix}(T_{P1}, T_{P2})$
    - **if** $\neg x_i = v_3$
      - **then**
        - $T_{P1} \leftarrow \text{extract}(T_P, NEG, v_1)$
        - $T_{N1} \leftarrow \text{extract}(T_P, POS, v_1)$
        - $T_{P2} \leftarrow \text{extract}(T_{N1}, NEG, v_2)$
        - $T_P \leftarrow \text{mix}(T_{P1}, T_{P2})$
- $T_P \leftarrow \text{append}(T_P, POS)$
- $T_N \leftarrow \text{append}(T_N, NEG)$
- $T \leftarrow \text{mix}(T_P, T_N)$

return (detect(T))

3.3 Implementations of molecular SATISFIABILITY solvers

In this section, we describe physical and simulated implementations for molecular SATISFIABILITY algorithms. This includes simulation of Lipton's and Ogihara and Ray's algorithms. We see a physical implementation of Ogihara and Ray's algorithm with manual laboratory procedures.

3.3.1 Physical implementations

Yoshida and Suyama implemented Ogihara and Ray's algorithm with manual molecular biology techniques [28]. This experiment solved a 3-CNF instance with four variables and 10 clauses.

3.3.2 Simulations

Martn-Mateos et al. introduced a simulation for Lipton's algorithm [19]. Molecular operations get implemented with ACL2, a Common Lisp variant. The framework for this environment implemented test cases for Lipton's algorithm.

Ogihara provides test results for implementation of his original molecular algorithm [20]. This simulation provides a comparison with 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 parses an input CNF expression into growing and self regulated set of possible combinations.

4.1 Distribution algorithm for SATISFIABILITY

The distribution algorithm parses an input CNF expression into growing and self regulated set of possible combinations. A possible combination begins with all members of the first clause. Variables get inserted into an expanding set of valid assignments. A clause gets eliminated when an assignment contains a conflict.

4.1.1 Description of the Distribution algorithm

Initially the algorithm starts with the variable assignments of a clause. Evaluation of subsequent clauses extends the solution space with the INSERT VARIABLE subroutine. During each insertion, the variable gets inserted into a potential solution vector. Table 4.1 lists the four possibilities for variable assignment.

Table 4.1: Configurations for the INSERT VARIABLE subroutine

Item	Return state	State
1	$v \cdot s$	if v is less than all elements in s
2	$s \cdot v$	if v is greater than all elements in s
3	$s_1 \cdot v \cdot s_2$	if v is between two elements in s
4	$\emptyset$	if v conflicts with $-v$ in s

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

A variable conflict occurs when both positive and negative assignments of a variable occur in a sequence s. In this case, the sequence s gets removed from the set potential solutions.

Redundant vectors get removed after insertion of the next disjunctive clause. Any remaining valuations in the solution space contain non-conflicting variable assignments. This does not immediately require that each valuation be a complete satisfiable assignment. Satisfiable valuations remain in a non-empty satisfying solution space.

Vectors that are of equal magnitude of the number of variables in the problem instance are satisfiable valuations. However, there may exist solutions that span only the required satisfiable assignments; that is activate each of the independent clauses with at least one non-conflicting assignment. This assignment may be the minimum valuation for the expression, in the case that the backbone consists of the variables of the maximum valuation.

4.1.2 Pseudocode for Distribution algorithm

Algorithms 4.1.1 and 4.1.2 provide pseudocode for the Distribution algorithm.

Algorithm 4.1.1: INSERT VARIABLE(v, s)

```
if s = ∅
  then T_R ← {v}
  else if v < s
  then T_R ← append(v, s)
  else if v > s
  then T_R ← append(s, v)
  else { Find the position i for v in s
         if v = s_i
           then { if v polarity is s_i polarity, T_R ← {s}
                  else T_R ← ∅
                  else if s_i > v
                  then { [s', s''] ← splice(s, s_i)
                         s' ← append(s', v)
                         T_R ← append(s', s'') }
return (T_R)
```

Algorithm 4.1.2: DISTRIBUTION SAT(ϕ)

```
m number of clauses
k number of variables in each clause

Initialize with the variables from the first clause
T ← {C_1}
for i ← 2 to m
  do { for j ← 1 to k
         do { T_j ← T
              x_j ← C_{i,j}
              for each string s in T_j
                do T_j ← mix(T_j, INSERT VARIABLE(x_j, s)) }
       for j ← 1 to k
         do T ← mix(T, T_j)
return (detect(T))
```

Chapter 5

Molecular Simulation: An environment for molecular computation

This chapter introduces Molecular Simulation: An environment 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 automated documentation, 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. Environment 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.

Molecular Simulation can be downloaded from `http://www.cs.rit.edu/~dnc6813/project/project.html`. The archive contains example DIMACS CNF testbench and ex-

ample instances. This document and the online documentation may be used independently for getting started.

5.2 Requirements

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

5.2.1 Hardware requirements

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

5.2.2 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.

5.3 Documentation

The simulation environment allows the user to execute DIMACS CNF input. Usage for algorithm implementation permits the user to design and test new molecular algorithms.

Detailed documentation can be accessed from the project website. This includes an overview of Molecular Simulation along with detailed class and function documentation. Any modifications to this software should use this as a starting point in development.

5.4 Tools

This project uses several tools for automating tasks and execution. In this section, we describe the tools for documentation, automated execution and output visualization for Molecular Simulation.

5.4.1 Doxygen

`Doxygen` can be used to generate automated documentation for Molecular Simulation. The online and offline documentation get generated from Doxygen formatted documentation. Download and learn `Doxygen` at `http://www.stack.nl/~dimitri/doxygen/`.

5.4.2 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.

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 expressions 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`

5.4.3 Data Visualization

We adopt a modified plot for visualization of output data. Ben Fry's example in Chapter 4 of Visualizing Data[9] provides a framework for importing output from Molecular Simulation. We provide the modified Processing source in the Molecular Simulation archive. The visualization directory contains a README for usage.

5.5 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
```

5.6 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 UNKNOWN
c
c A line beginning with a ‘s’ marks the status.
c This can be either ‘UNSATISFIABLE’, ‘SATISFIABLE’, or ‘UNKNOWN’.
c
v 0
c
c A satisfiable valuation begins with a ‘v’ and ends with a ‘0’.
c     Between the ‘v’ and ‘0’ is a sequence of integers encoding the valuation.
```

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 expression.
`c numClause:`	Display the number of `clauses` in the input CNF expression.

5.7 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`	`[output]`	Write output to file Output filename

Let us consider an 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 with Molecular Simulation. Molecular Simulation requires a 64-bit architecture with a UNIX like environment 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 with 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.

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

Once the execution has completed for all test instances, we can analyze the output. This 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. With the `-f` option, output may be saved to a file. With the `-f` option specified, output get saved with `[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 and discuss the design decisions for a general purpose molecular computer.

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.

Design considerations for a physical molecular computation environment extend the algorithm test results. Selection of a molecular SATISFIABILITY algorithm considers the simulation results and practical implementation techniques.

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 expression.

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 expression.

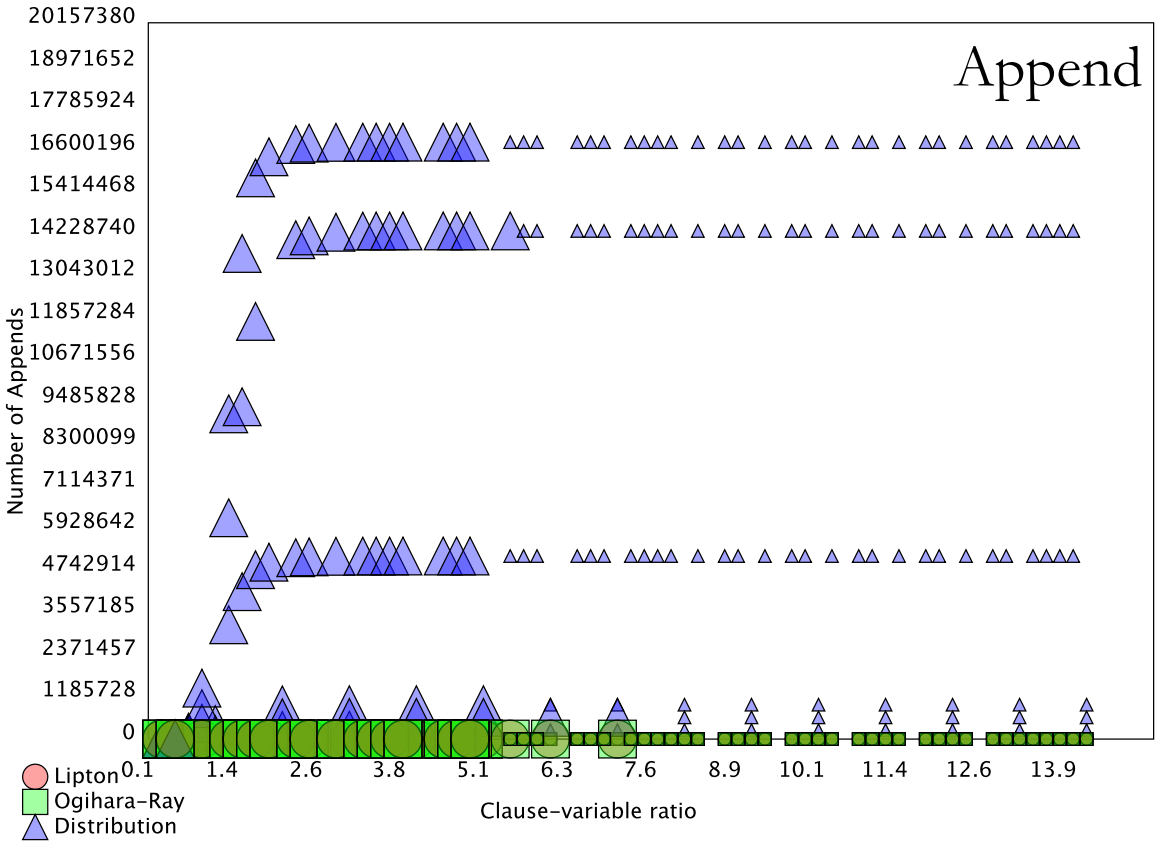

Figure 7.1: Clause to variable ratio α vs. Number of appends

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.

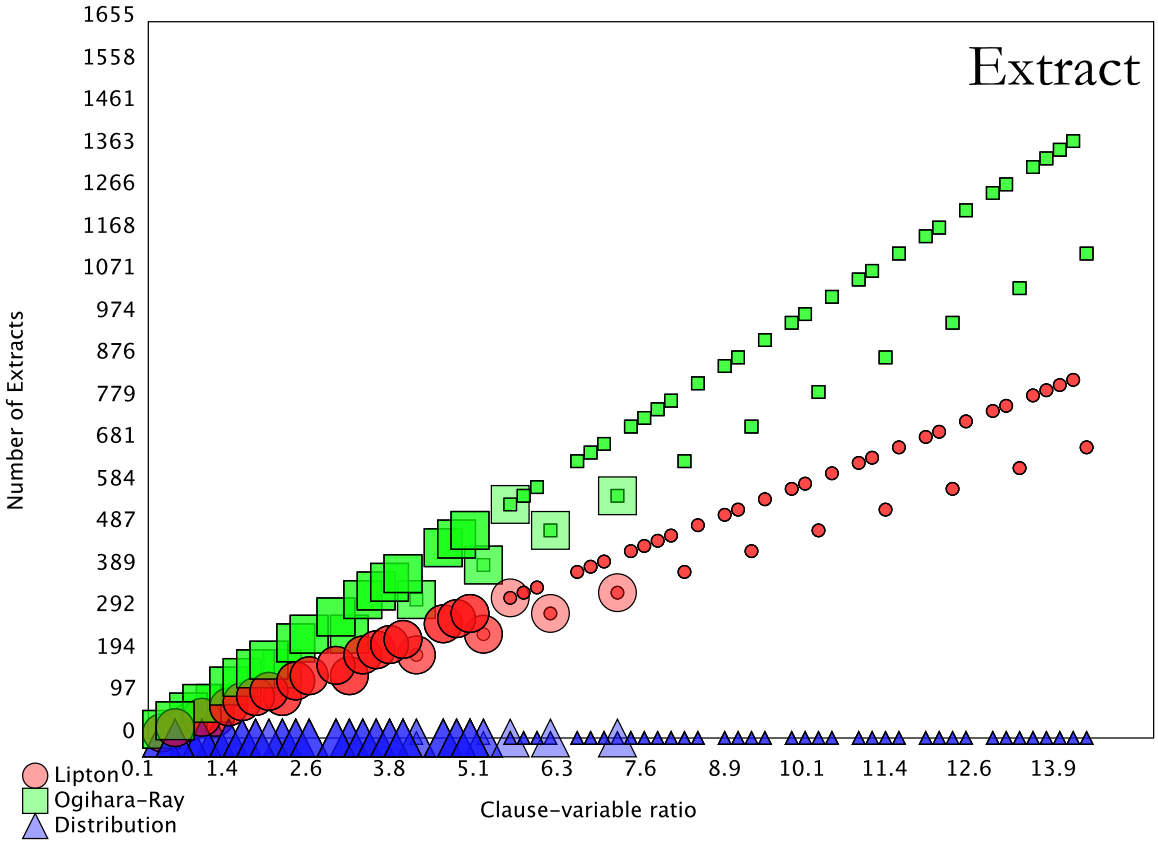

Figure 7.2: Clause to variable ratio α vs. Number of extracts

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.

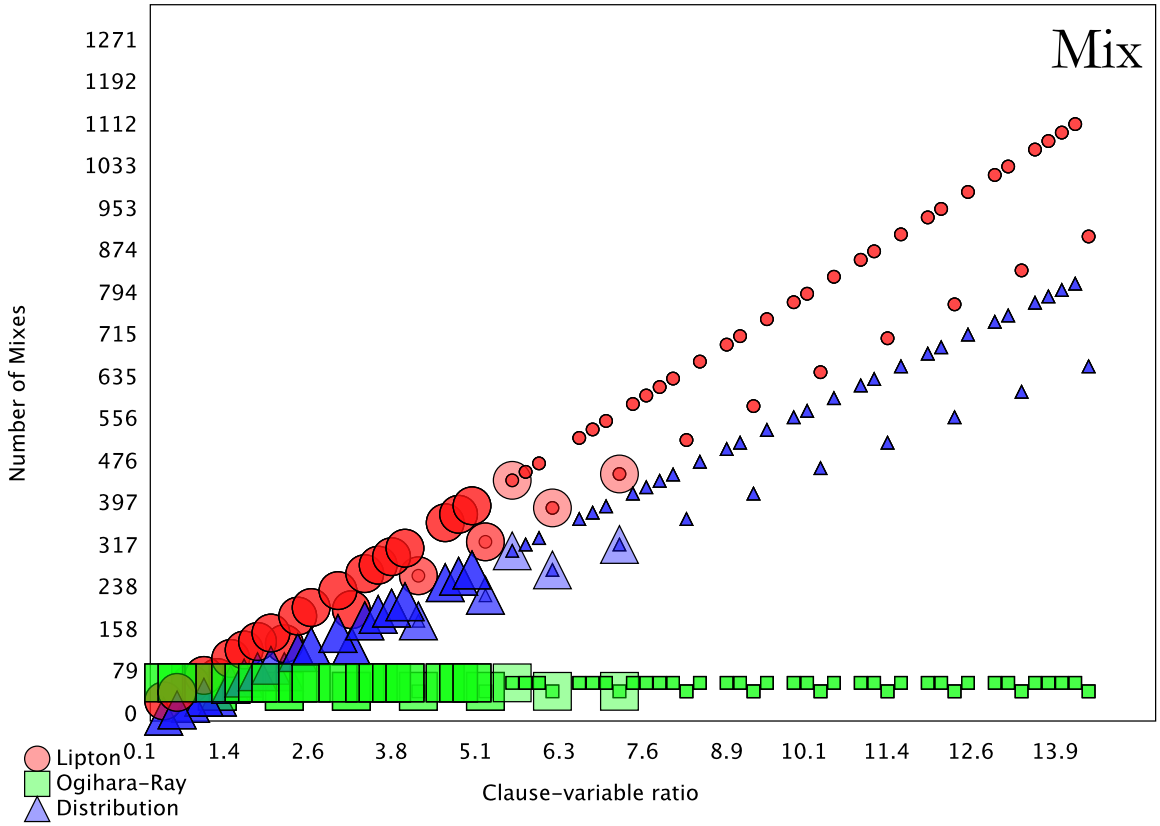

Figure 7.3: Clause to variable ratio α vs. Number of mixes

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

All three algorithms operate with 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 with Lipton's and the Distribution algorithms.

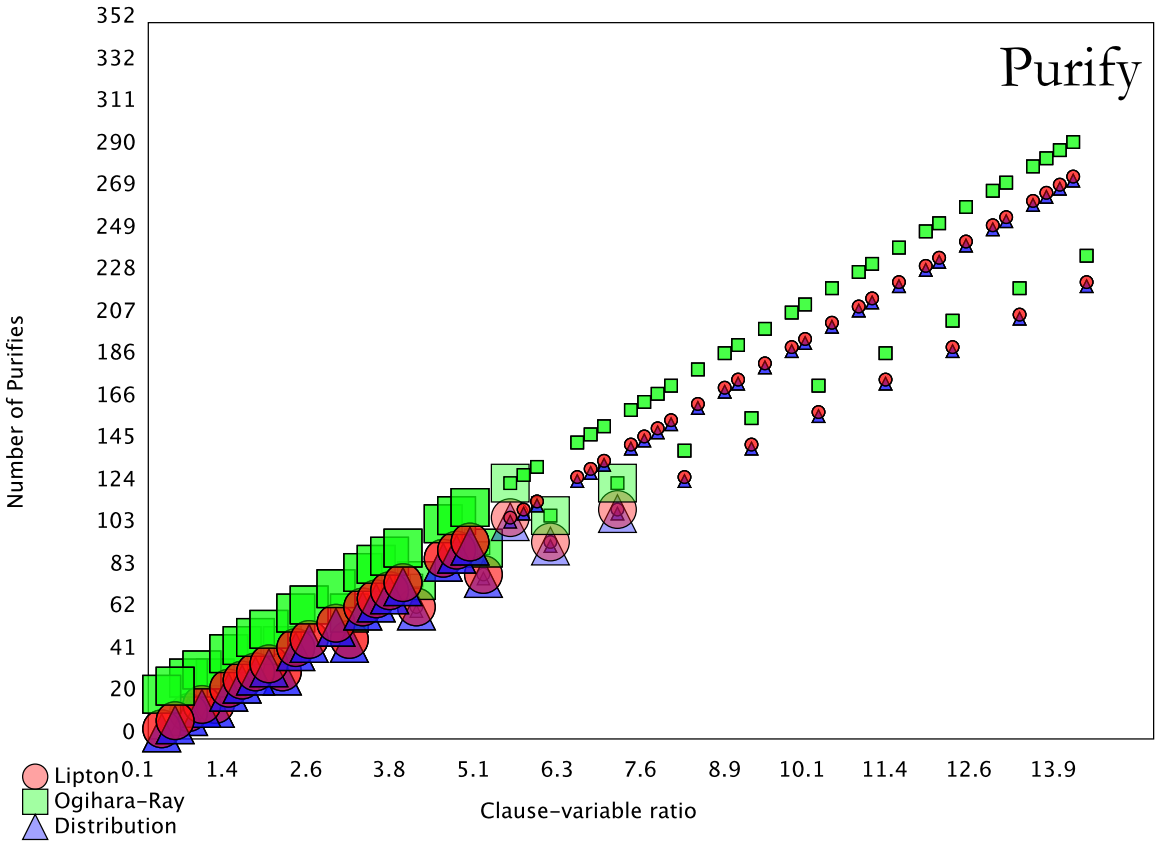

Figure 7.4: Clause to variable ratio α vs. Number of purifies

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 expression. Each split requires reassembly, accomplished with two appends. Figure 7.1 shows the number of appends.

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

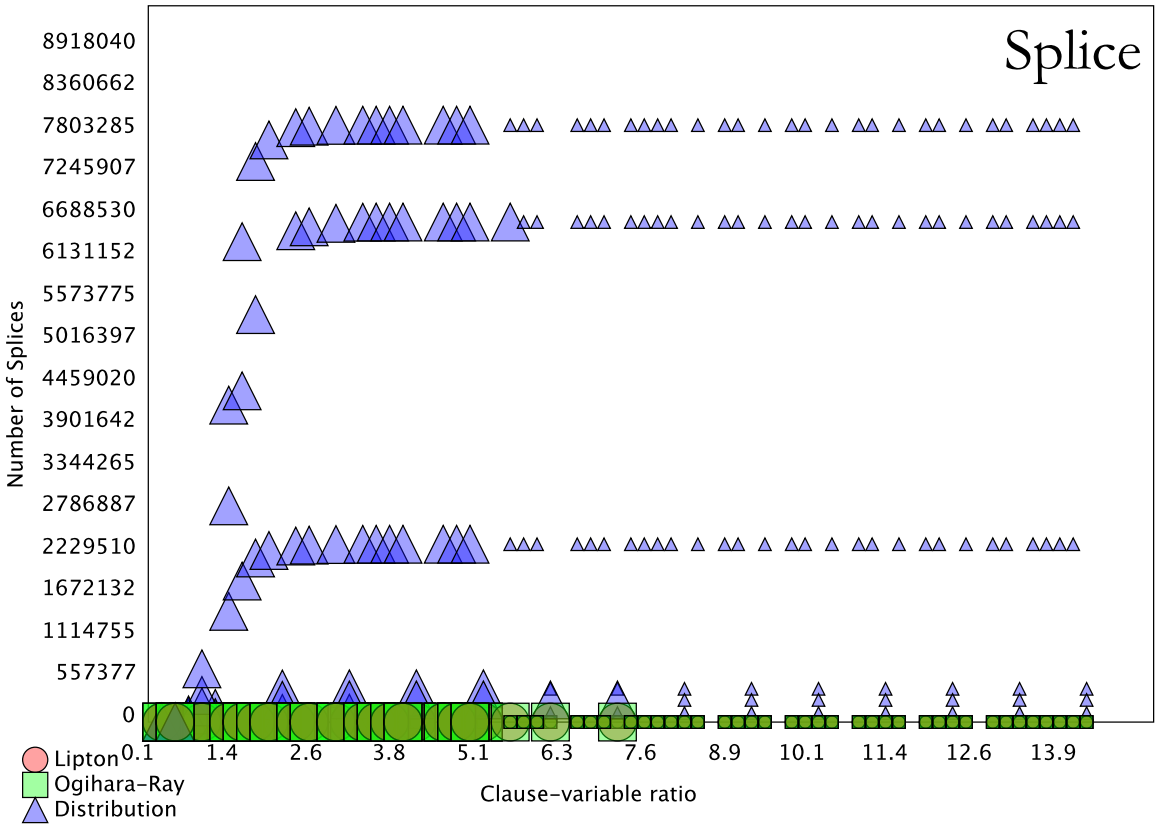

Figure 7.5: Clause to variable ratio α vs. Number of splices

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.

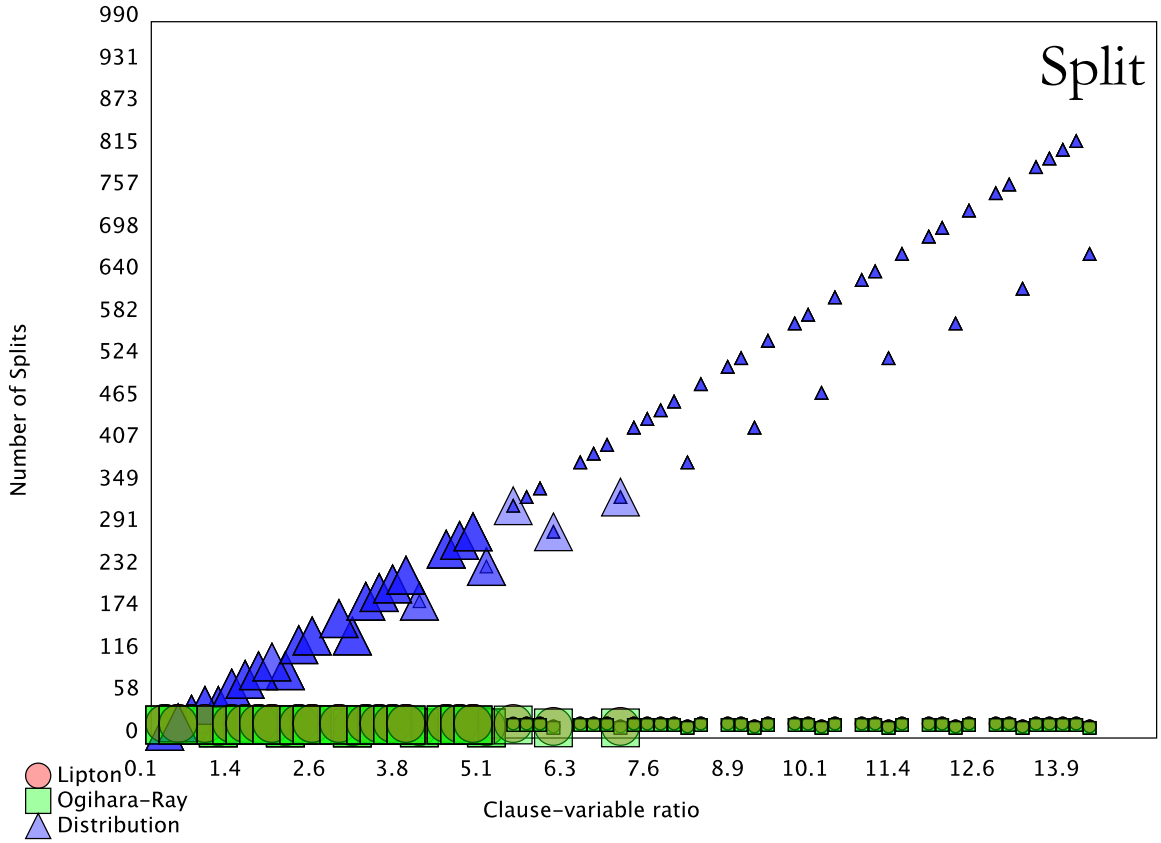

Figure 7.6: Clause to variable ratio α vs. Number of splits

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 with $\alpha \approx [4.2, 8.2]$ taking the longest. This is within the phase-transition region for 3-SAT.

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.

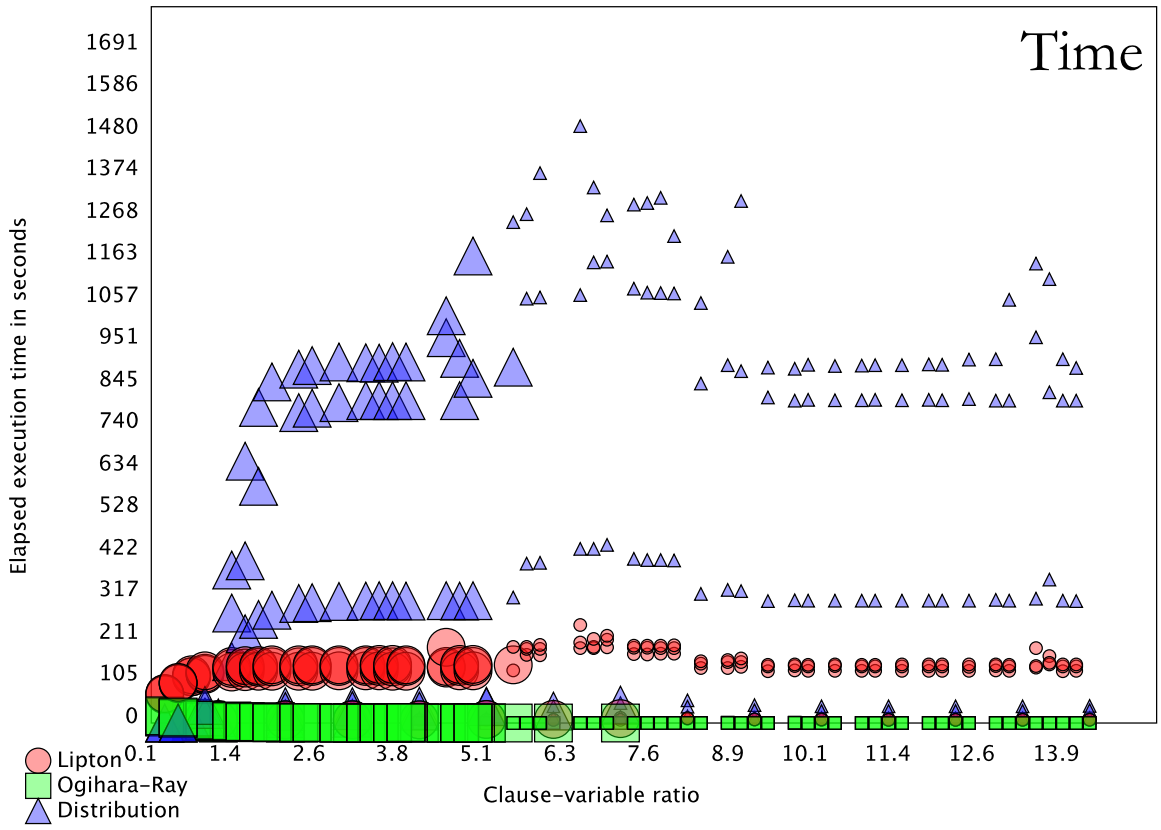

Figure 7.7: Clause to variable ratio α vs. execution time in seconds

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 valuations for only necessary variables.

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.

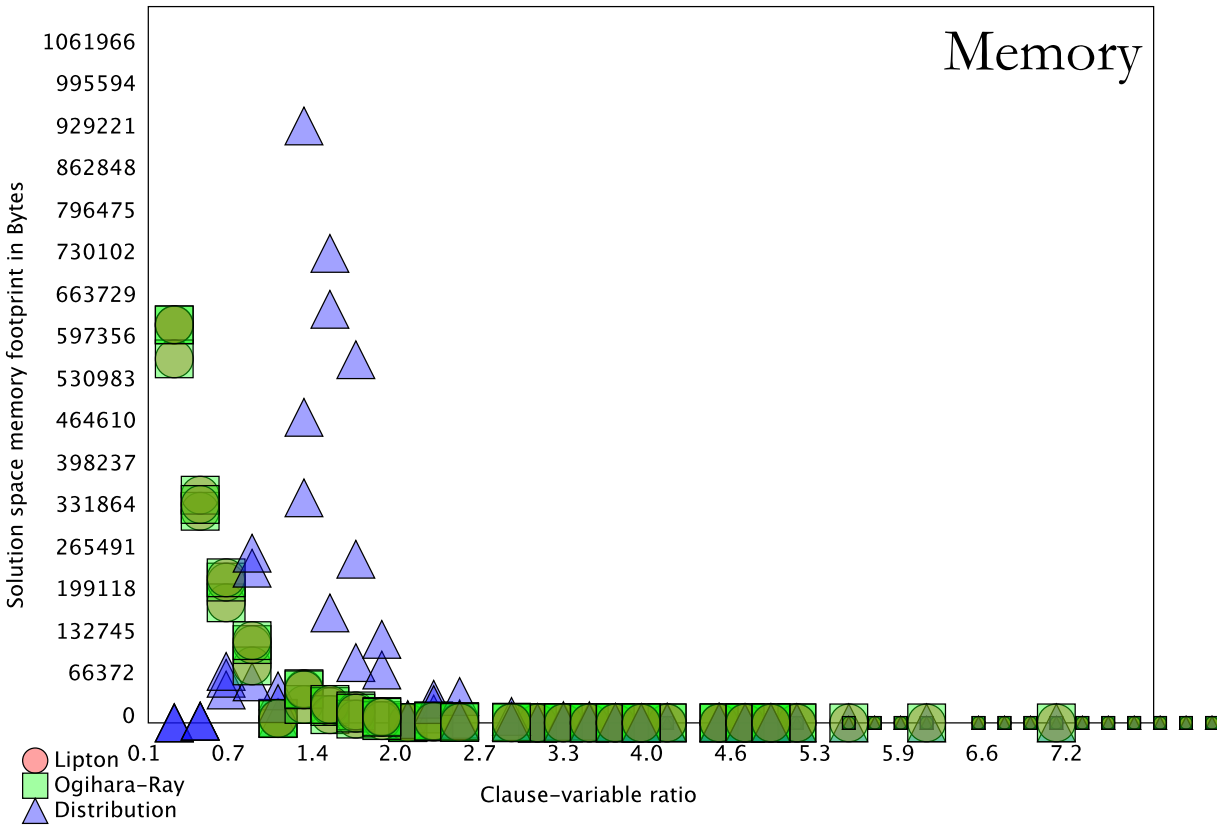

Figure 7.8: Clause to variable ratio α vs. satisfiable solution footprint in Bytes

7.2 On construction of a molecular computer

We consider present techniques in molecular biology and VLSI circuit construction as design considerations for a molecular computer. We describe the selection of an encoding substrate and for sequence representation. Implementation of the substrate considers properties from the molecular algorithms under test.

Continuing, we provide techniques for construction of a molecular computation environment. The environment applies SATISFIABILITY as a standard algorithm for general computation. This includes an execution pipeline for integration with existing silicon fabrication processes.

7.2.1 Selection of encoding substrate

During the introduction of molecular computation, we considered +DNA and −DNA as a computing substrate. These choices were arbitrary to illustrate the automatic string matching mechanism of genetic information. Physical implementations can use various substrates for artificial or practical significance.

Designing real molecules with artificial significance require consideration of the practical implications. Physical features inadvertently encoded in a sequence may yield unexpected consequences. This includes molecules with structural design flaws or potentially hazardous.

Constraining molecular computation reduces the risk of external contamination. We consider an environment constructed for isolation. This eliminates external contamination of the molecular substrate and reduces the risk for contaminating the environment.

7.2.2 Selection of algorithm

Molecular Simulation provides a simulation environment for molecular SATISFIABILITY algorithms. We have collected simulation results for a sweep of random k-SAT instances. Now we consider each of the algorithm's performance in terms of implementation in a physical environment.

We have seen that some of the metrics vary considerably. Beneficial components of algorithm performance should be considered for physical implementation. This includes the benefits of reducing an existing space versus construction of only possible combinations.

Construction of a computational environment depends on the algorithm implementation. Lipton's algorithm reduces a larger space to only the satisfiable solutions. Ogihara and Ray's and the Distribution algorithm construct an incremental space based on the evaluation order. The incremental construction decreases the total amount of required space for computing SATISFIABILITY. With the Distribution algorithm, non-conflicting assignments provide shorter valuations without spanning all n variables.

7.2.3 Selection of encoding mechanism

The encoding mechanism may exhibit features that can be utilized for desired functionality. We consider construction and reuse of combinatorial spaces, along with permission of unbounded length restrictions.

Lipton's algorithm begins with the construction of a combinatorial space. The algorithm discards the space after each SATISFIABILITY instance. This does not need to be the case; the space can be recovered through a reaction process. The process requires copying the discarded +DNA strands and from the satisfying solution.

Let us consider a molecular computer without a fixed length bound. This machine is the same as the construction of a combinatorial space without regarding the a fixed number of variables. With this machine, we can provide assertions for problems beyond a fixed problem instance. Immediately we can represent much larger problems that are representable in the current vector space. This leads to a possible security concern that may allow the user to increase the space for solving any problem. The present oligonucleotide definition does not provide efficient encodings compared with natural genomes.

7.2.4 Description of a self contained molecular computer

Manual laboratory procedures demonstrate molecular computation for single problem instances [1, 28, 7, 3]. These techniques are error prone due to human intervention. Implementers have suggested robotic automation as a means to eliminate human contact [28]. However the techniques are an automated version of a human process. We consider present gene sequencing technologies for the construction of a general molecular computer.

Integrated molecular computing technology gets inherited from sequencing technologies. Nanopores provide a method for sequencing genetic sequences [18, 23, 16, 22]. We apply nanopores along with integrated micropumps [15] for the transfer of molecules within an integrated architecture.

Construction of the integrated architecture extends standard silicon processes. A nanopore can be constructed by production of a silicon nitride (Si_3N_4) substrate and removing an hourglass shaped portion [13]. Each side of the nanopore forms a membrane between two potentials [13]; the potential difference requires a concentration of potassium (K^+) and chlorine (C^-) ions in aqueous buffer solutions.

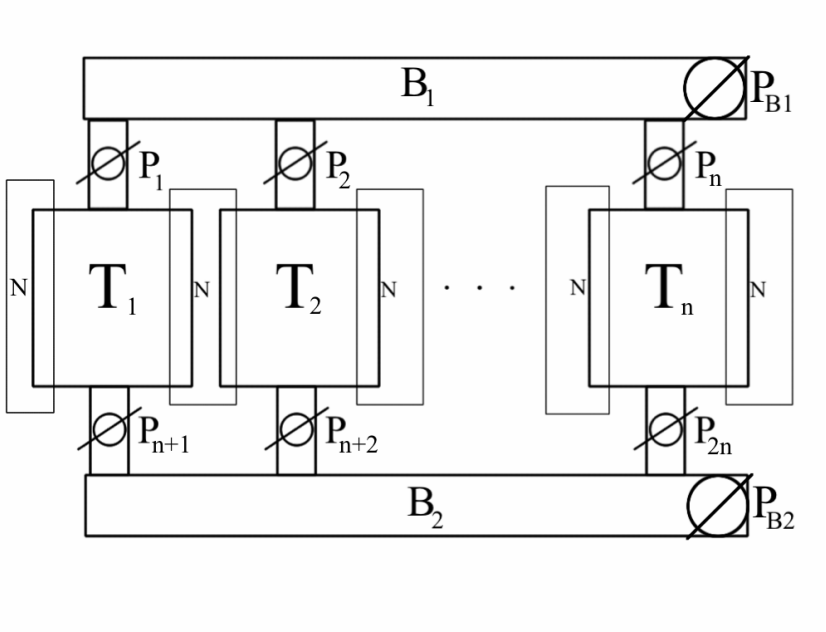

Figure 7.9: Schematic of molecular computing architecture.

Figure 7.9 shows a schematic representation for a molecular computing environment. This environment consists of a series of tubes connected with nanopores (N) and micropumps (P). Each well (T) contain the representation of a test tube in a macro-experimental view. Information between wells gets transfered through a stackable microtube data bus, labeled B_1 and B_2. The microtube data bus provides transfer of buffer solution and preparation of wells.

Control of the machine in Figure 7.9 requires the transfer of buffer solutions along with molecular encodings between tubes. Activating a micropump permits flow through the tube and directed movement of solutions through an internal configuration. For example, directing the pumps P_1 and P_{n+1} from B_1 to B_2 permit the contents of T_1 to be mixed with B_2. Once this transfer occurs, then P_{n+1} can be closed. Subsequent operations may be performed with multiple tubes, and the contents directed to specified storage.

Nanopores allow for reading and writing of genetic sequences. In the molecular computing architecture the read and write cycle occurs in a isolated tube. As a molecule passes through the nanopore, a current is induced according to the nucleotide present in the pore. We cannot yet practically write molecules with nanopores. Writing with nanopores requires modification of a molecular state, and can theoretically be accomplished [18]. The nanopore will consist of a full read-write component and provide an interface for external control.

Integration with existing computing architectures provides an interface to the user and a synergetic medium for general purpose computation. Molecular computing can perform many tasks in parallel, however each operation may take orders of magnitude longer than conventional architectures. For example, a conventional computer architecture may be used to verify valid configurations for SATISFIABILITY but cannot easily compute all valid configurations. The computation of all configurations for a SATISFIABILITY problem instance may be executed as a combinatorial molecular process, and the solution written to a conventional computing medium. This interface requires design and interaction with the user and can leverage computer architectures as a dual environment.

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 conventional computing environment. 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 computing during this project. This includes introducing the molecular 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.

Finally, in Chapter 7, we provided the experimental results and an overview of state-of-the-art gene sequencing technologies. With these technologies, we introduced a generalized molecular computation architecture. This architecture shares techniques employed in current gene sequencing tools.

8.2 Future work

Nanopore sequencers have been designed for reading molecules and diagnosing patients in a medical setting. Creation of molecular computation architectures permit generalized computation with physical encodings.

Sequencing for medical purposes will continue to drive innovation. Construction of the molecular architechure provides tools beyond general purpose computing. The computing architecture operates as a physical molecular laboratory. This provides the ability to replicate molecular interactions in a controlled setting and observe real interactions of genetics.

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] Faulhammer, D., Cukras, A., Lipton, R., and Landweber, L. Molecular computation: RNA solutions to chess problems. *Proc Natl Acad Sci U S A 97*, 4 (2000), 1385–9.

[8] 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).

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

[10] 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).

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

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

[13] Heng, J. B., Aksimentiev, A., Ho, C., Marks, P., Grinkova, Y. V., Sligar, S., Schulten, K., and Timp, G. The electromechanics of DNA in a synthetic nanopore. *Biophysical Journal 90*, 3 (2006), 1098–106.

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

[15] Liao, C.-S., Lee, G.-B., Liu, H.-S., Hsieh, T.-M., and Luo, C.-H. Miniature RT–PCR system for diagnosis of RNA-based viruses. *Nucleic Acids Research 33*, 18 (2005), e156.

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

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

[18] 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.

[19] 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.

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

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

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

[23] Peng, H., Luan, B., and Stolovitzky, G. *Nanopores: Sensing and Fundamental Biological Interactions*. Springer, 2011, ch. 11.

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

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

[26] 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.

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

[28] 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.

[29] 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.