Volume Bounded Molecular Computation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

By

Bin Fu

September 15, 1997

Abstract

The number of molecular strands used by a molecular algorithm is an important measure of the algorithm's complexity. This measure is also called the *volume* used by the algorithm. In this thesis, we study volume bounded molecular computation in which the number of operations and the length of all DNA sequences are bounded by a polynomial in the length of the input.

We show that every NP problem that can be solved with $b(n)$ bits of nondeterminism can be solved by molecular computation with four test tubes, in volume $2^{b(n)}$. In addition, we identify a large class of recursive algorithms that can be implemented using bounded nondeterminism. This yields improved molecular algorithms for important problems like 3-SAT, independent set, and 3-coloring.

We prove that three important polynomial-time models of molecular computation with bounded volume are equivalent to models of polynomial-time Turing machine computation with bounded nondeterminism. Without any assumption, we show that the Split operation does not increase the power of polynomial-time molecular computation. Assuming a plausible separation between Turing machine complexity classes, the Amplify operation does increase the power of polynomial-time molecular computation.

We develop a general technique for constructing molecular-based approximation algorithms for NP optimization problems. Our algorithms exhibit a useful volume–accuracy tradeoff. We also present the first candidate for a problem that can be solved more efficiently with the Amplify operation than without.

We show a method for comparing the number of DNA sequences in two test tubes via annealing. It induces a molecular algorithm to compute #SAT and a molecular computation with subexponential volume to improve the power of two-sides error random computation.

We design a molecular algorithm for the well known Hamiltonian path problem. Its volume complexity is 2^n which is in contrast to the exhaustive method that uses $n!$ volume of solution.

Acknowledgments

I would like to express my sincere gratitude to Richard Beigel, my advisor. He introduced me to molecular computation. I greatly appreciate his guidance and continuous support throughout this research. I also thank him for his comments during the preparation of the thesis.

It was very enjoyable to study at Yale as a graduate student. I am grateful to Dana Angluin, James Aspnes, Michael Fischer, and Laszlo Lovasz for creating a wonderful theory environment. James Aspnes made many valuable comments on an earlier version of this thesis. Special thanks go to faculty of the department of Computer Science at Yale for providing excellent program.

I would like to thank Anne Condon for serving on my thesis committee. Her comments on Chapter 6 greatly improve our presentation.

I would like to thank Eric Allender, Lane Hemaspaandra, Steven Homer, Ming Li, Mitsunori Ogihara, Jie Wang, and Andrew Yao for their help during my graduate study.

I would like to thank two important people in my academic career for their long lasting influence. Qiongzhang Li, my former advisor at Wuhan University, led me into theoretical computer science. Under his guidance, I built a solid foundation in theory of computation. Shouwen Tang, my colleague at Beijing Computer Institute, introduced me to complexity theory.

I am indebted to many colleagues at Yale university for their help and friendship. I particularly wish to thank Dongqi Liu, Jianxiang Liu, Hong Xiao and Dongpei Zhuo. I would also like to thank Fangxiao Zhou for his patience in helping me understand the biological process of DNA manipulation.

I would also like to thank William Gasarch and Zengyan Zhang for their help when I was visiting at University of Maryland.

I would like to thank my wife Feng Li for her constant encouragement.

Contents

Chapter 1.

Introduction

1.1. Computing with DNA

Molecular computation was first proposed by Feynman [22], but his idea was not implemented by experiment for a few decades. In 1994 Adleman [1] succeeded to practically solve an instance of the Hamiltonian path problem in a test tube, just by handling DNA strings. A directed graph with initial vertex s and target vertex t is said to have a Hamiltonian path if there is a directed path that begins at s, ends at t, and enters every other vertex exactly once. Determining if a directed graph has a Hamiltonian path is one of the most famous NP-complete problems. Adleman's experiment is a major milestone in molecular computation and opened the field of practical DNA computing. In recent years there have been a lot of papers contributing to DNA computation.

DNA is the storage medium for genetic information. It is found in every cellular organism. It is composed of units called nucleotides distinguished by the chemical group attached to them. The four bases are adenine, guanine, cytosine, and thymine, abbreviated as A, G, C and T. Single nucleotides are linked together end-to-end to form DNA strands. Every DNA strand has two distinct ends, one with a free $5'$ PO_4 group and the other with a free $3'$ OH group, called $5'$ and $3'$ ends, respectively. Thus, we can think of the sequence of nucleotides in any strand as having a natural orientation.

Mutual attraction, called hydrogen bonding, exists between As and Ts and between Gs and Cs. Hence, A/T and G/C are called complementary base pairs. For two complementary single strands, hydrogen bonding will only occur if the pair of complementary strands are oriented in an anti-parallel fashion. That is, as one strand of a double helix runs in the $5' \to 3'$ direction, its complement runs in the $3' \to 5'$ direction. Two complementary single strands can pair up and twist around each other, forming the famous double helix structure discovered by Watson and Crick. As DNA strands consist of four different symbols A, G, C, T, they can encode conventional strings in the conventional computer.

1.2. DNA Manipulations

In the last decade there have been revolutionary advances in the field of biomedical engineering particularly in recombinant DNA and RNA manipulating. Due to the industrialization of the biotechnology field, laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [58, 59]. In this section we describe some biological operations that are useful for molecular computation. Some of our descriptions are similar to those in [12, 31, 44].

For a string x we denote by $\uparrow x$ the single DNA strand whose sequence is x and whose orientation is $5' \to 3'$, as x is read from left to right. Similarly, $\downarrow x$ denotes the strand complementary to $\uparrow x$. Finally, $\updownarrow x$ denotes the double strand that results when $\uparrow x$ and $\downarrow x$ anneal.

1.2.1. Merge

This is to pour the solution of two test tubes into a third one to achieve union. It can be performed by re-hydrating the tube contents and then combining the fluids together into a new tube by pouring.

1.2.2. Melting

This is to break double DNA strands into its single-stranded complementary components by heating the solution.

1.2.3. Separation by Length

This is to separate all DNA sequences by their lengths. A technique called gel electrophoresis is used. The molecules are placed at the top of a wet gel, to which an electric field is applied, drawing them to bottom. Larger molecules travel more slowly through. After a period, the molecules spread out into distinct bands according to size.

1.2.4. Separation by Subsequence

This is to extract all strands containing the sequence $\uparrow x$. We use the method of biotin–avidin purification as described in [1]. We first create many copies of $\uparrow x$'s complementary oligo (a short DNA strand), namely $\downarrow x$. To these oligos we attach biotin molecules, which are in turn anchored to an avidin bead matrix. We then melt the double strands in our test tube and pour them over this matrix. Those strings that contain $\uparrow x$ anneal to the $\downarrow x$ oligos anchored to the matrix. We wash away all strands that did not anneal, leaving behind only those strands that contain $\uparrow x$, which can then retrieved from the matrix.

1.2.5. Amplification (PCR)

This is to make copies of DNA strands by using the polymerase chain reaction (PCR). If we have the duplex $\updownarrow xyz$, we first melt it to form $\uparrow xyz$ and $\downarrow xyz$. To this solution we add the primer oligos $\downarrow z$ and $\uparrow x$, which anneal to form the partial duplexes $\uparrow xy \updownarrow z$ and $\updownarrow x \downarrow yz$. DNA polymerase can then elongate the primer to create full duplexes of the form $\updownarrow xyz$. Now we have two copies of our original strand. Thus, if we can guarantee that the primer sequences that we use occur on the ends of every strand, and only on the ends, then we can use PCR to duplicate every strand in the test tube.

1.2.6. Append

This is to elongate every strand in a test tube T by tacking another short strand z onto the end. Suppose $z = D_1 D_2 \cdots D_k$, where each $D_i \in \{A, T, G, C\}$. A solution containing D_1 is poured into T, and each sequence d reacts with D_1 to form dD_1.

After washing the excess D_1 solution away, one could have D_1 from the chain dD_1 coupled with D_2 to form dD_1D_2, and so on. Each character is appended in the $3'$ end (It will be not appended at $5'$-end). Each time only one character is appended. This can be guaranteed by the biological techniques that let the new character be in protected state. The protected state can be changed into unprotected state so that another character can be appended.

1.2.7. Annealing

This is to bond two single-stranded partially complementary DNA sequences by cooling the solution. $\uparrow\alpha\sigma$ and $\downarrow\sigma\beta$ will be joined at $\uparrow\sigma, \downarrow\sigma$ when two strands meet in the cooled solution.

1.2.8. Detect

This is to test if there is at least one DNA sequence in the solution of a test tube. It is usually done by amplification (see [49]).

1.3. Adleman's Experiment for Hamiltonian Path Problem

In this section we give a brief description of Adleman's [1] experiment for solving Hamiltonian path problem. Similar descriptions can be also found in [54, 31]. The input is a graph with vertices V and edges E. The output is yes if there is a Hamiltonian path; no, otherwise.

1.3.1. Algorithm

i. Generate random paths through the graph.

ii. Keep only those paths that begin with s and end with t.

iii. Keep only those paths that enter exactly n vertices.

iv. Keep only those paths that enter all of the vertices of the graph.

v. If any paths remain, return yes; otherwise, return no.

1.3.2. Biological Implementation

i. Choose at random n distinct single-stranded 20-character DNA sequences, and to each vertex v associate sequence S_v. For each such sequence obtain its reverse complement $\overline{S}_v$. Generate many copies of each $\overline{S}_v$ in test tube T_1.

   If $(u, v) \in E$, then build sequence S_{uv} by concatenating the 10-base suffix of S_u to the 10-base of S_v. Generate many copies of each S_{uv} sequence in test tube T_2.

   Pour T_1 and T_2 into T_3. We assume that in T_3 any binding between can freely take place. It forms all of the possible paths.

   For a path $u_1u_2\cdots u_k$, let's consider how it can be formed via the above operations. A 10-base suffix of one S_{u_1} will bind to the 10-base prefix of one $S_{u_1u_2}$ because one is complementary to the other; at the same time the 10-base suffix of the same $S_{u_1u_2}$ binds to the 10-base prefix of one S_{u_2} sequence whose 10-base suffix binds to the 10-base of one $S_{u_2u_3}$, and so on (see Section 1.2.7).

ii. Solution of (i) is amplified by polymerase chain reaction (see Section 1.2.5). In the process those molecular encoding paths that began with s and ended with t will be amplified. The primers are S_u and $\overline{S}_v$.

iii. Separate the double-stranded DNA having exactly $20n$ bases, which therefore represent paths with exactly n vertices. Apply the gel electrophoresis techniques of Section 1.2.3 which can separate DNA sequences by length.

iv. Use affinity purification. Filter out all of the strands where each S_u is present for each $u \in V$ (see Section 1.2.4).

v. Amplify the result of (iv) by polymerase chain reaction (see Section 1.2.5) and then determining the DNA sequence of the amplified molecular.

1.4. Limitations of Molecular Computation

Although molecular computation seems have certain advantages over conventional computer, it is severely limited by the current techniques in DNA manipulation. We list some of them that we believe to be important.

1.4.1. Volume of Solution

This is the number of elements in a test tube. It was estimated by Adleman that a liter of solution has 2^{70} DNA strands. Volume should not exceed 2^{70}. This limits the size of NP search problem. Volume may be the most important limitation of molecular computation. It is also the main topic of this thesis.

1.4.2. Time Duration of Each Lab Step

Merge of test tubes by mixing can be done in a few seconds. The time of operations (such as Separation, Annealing) depending on complementary matching of DNA sequences is strongly dependent on the length of the required match and the allowed number of errors (and may also depend on temperature and other parameters).

1.4.3. Number of Lab Steps

Due to the extremely slow steps rate of the lab operations, the number of steps should be small.

1.4.4. Length of DNA Sequences

Operations on short DNA sequences are easier than on long sequences. Long DNA sequences bring much inconvenience for the operations.

1.4.5. Error of Operations

The biological operations are not free from error. They are affected by temperature and by ionic strength of the solvent. It is not perfectly clear to what accuracy a biological operation can be performed. Some people have investigated how to cope with certain errors that may arise in molecular computations [14, 44, 32, 18].

1.4.6. Related Work

Following Adleman's invention, Lipton [36] designed a simple molecular algorithm for finding solutions to the SAT problem. It is the first NP-complete problem discovered by Cook [19]. The SAT problem is to determine if a boolean formula has a truth assignment that makes the formula true.

Both Adleman and Lipton used exhaustive searching in solving the Hamiltonian path problem and the SAT problem. In Adleman's algorithm $n!$ DNA strands are required to encode all paths. In Lipton's algorithm, 2^n DNA strands are needed to encode all of the truth assignments for the n variables. If we can use at most 2^{70} DNA strands, their algorithms can only deal with inputs of small size. Hartmanis [27] showed that a mass of DNA greater than that of the earth would be required to solve a 200-vertex instance of the Hamiltonian path problem. Solving NP-complete problems with molecular computation was further studied in a series of papers [13, 3, 39, 9, 48, 38, 40]. In [3, 39, 9, 38], efforts were made in optimize the volume of DNA solution.

Applying a molecular computer to solving various computational problems was studied in a series of papers [48, 5, 44, 12]. In [12], a molecular algorithm was designed for breaking the Data Encryption Standard (DES). Breaking DES means that given one (plain-text, cipher-text) pair, we can find a key which maps the plain-text to the cipher. DES is widely used in the modern society. There are 2^{56} cases for searching for a key for DES. No current computer can do it via exhaustive searching. It is estimated that DES could be broken in 4 months by using molecular computation.

In [2, 5, 47, 48, 60, 62], researches were taken to simulate a Turing machine by a DNA model of computation that uses biological operations. Those works may serve as the theoretical basis for eventually constructing a practical molecular computer.

The problems mentioned above that molecular computation has the potential to outperform existing computers. One of the reasons is that the operations that molecular biology currently provides can be used to organize massively parallel searches.

The final goal of this field is to build a molecular computer that is more powerful than classical computer in solving some important problems. One aspect of the research is to make the biological operations practical. and look for new operations for DNA that will be useful for computation. Another aspect is to find a killer application. That is, an application of a molecular computer that would establish its superiority over others in a particular domain. An Assured future for DNA computation can only be established through the discovery of such an application.

DNA computing is still at its early stage. While we do not believe that DNA computing can eventually replace conventional computing, it may be more efficient in solving some problems that are intractable for a conventional computer.

1.5. Our Contributions

We show that every NP problem that can be solved with $b(n)$ bits of nondeterminism can be solved by molecular computation in a polynomial number of steps, with four test tubes, in volume $2^{b(n)}$. In addition, we identify a large class of recursive algorithms that can be implemented using bounded nondeterminism. This yields improved molecular algorithms for important problems like 3-SAT, independent set, and 3-coloring. These results were first presented in [9].

In Chapter 4, we relate molecular complexity to classical complexity. In particulars, we prove that three important polynomial-time models of molecular computation with bounded volume are equivalent to models of polynomial-time Turing machine computation with bounded nondeterminism. Without any assumption, we show that the Split operation does not increase the power of polynomial-time molecular computation. Assuming a plausible separation between Turing machine complexity classes, the Amplify operation does increase the power of polynomial-time molecular computation. These results were first presented in [23].

In Chapter 5, we develop a general technique for constructing molecular approximation algorithms for NP optimization problems. Our algorithms exhibit a useful volume–accuracy tradeoff. In particular we solve the Covering problem of Hochbaum and Maass using polynomial time and $O\left(\ell^2(\log \ell)n^2\binom{n(n-1)}{\ell^2/2}\right)$ volume with error ratio $(1+\frac{1}{\ell})^2$. We also present the first candidate for a problem that can be solved more efficiently with the Amplify operation than without. These results were first presented in [24].

In Chapter 6, we design a molecular algorithm to compute #SAT. We show how to use molecular computation in polynomial time and subexponential volume to improve the power of random computation. All operations used are practical in biology.

In Chapter 7, we design a molecular algorithm for the well known Hamiltonian path problem. Its time complexity is bounded by a polynomial and its volume is 2^n. This is in contrast to the exhaustive method that uses $n!$ volume.

Chapter 2.

Definitions and Background

2.1. Molecular Computation Models

The models we define were inspired by the work of [3, 46]. A molecular sequence is a string over an alphabet Σ (we can use any alphabet we like, encoding characters of Σ by finite sequences of base pairs). A test tube is a multiset of molecular sequences. We describe the allowable operations below. Where set notation is applied to multisets, multiplicities are respected. In the definitions, T_1, T_2, and T_3 denote distinct test tubes and c denotes a character.

The running time for a molecular algorithm is proportional to the number of operations on test tubes. An important complexity measure is the "solution volume size," *i.e.*, the maximum number of strings in a tube, counting multiplicities. Adleman [2] has speculated that molecular computation with a solution volume of size 2^{70} might be possible.

2.1.1. Encoding

Let $x = x_1 \cdots x_n$ be an n-bit binary string. Assign a unique sequence of 30 bases to each bit position and bit value. Sequence $B_i(0)$ encodes the ith bit of x as 0. Sequence $B_i(1)$ encodes the ith bit of x as 1. $x = x_1 \cdots x_n$ is represented by the following DNA sequence:

$$B_1(x_1)B_2(x_2)\cdots B_n(x_n).$$

We require that for any $\langle i, a\rangle \neq \langle j, b\rangle$, $B_i(a)$ and $B_j(b)$ have no long common subsequence.

2.1.2. Operations

We list some operations that we will study in the next three chapters. We will introduce some other operations in Chapters 6 and 7.

Sep(T_1, c, T_2, T_3)

$T_2 :=$ the multiset of all strings in T_1 that contain the character c anywhere;

$T_3 :=$ the multiset of all strings in T_1 that do not contain the character c anywhere;

$T_1 := \emptyset$.

Separate(T_1, c, i, T_2, T_3)

$T_2 :=$ the multi-set of all strings in T_1 whose ith character is c;

$T_3 :=$ the multi-set of all strings in T_1 whose ith character is not c or length is smaller than i;

$T_1 := \emptyset$.

Pour(T_1, T_2)

$T_2 := T_1$;

$T_1 := \emptyset$.

Append(T, c)

$T := \{xc : x \in T\}$.

Merge(T_1, T_2, T_3)

$T_3 := T_1 \cup T_2$;

$T_1 := \emptyset$;

$T_2 := \emptyset$.

Split(T_1, T_2, T_3)

T_2 and T_3 are obtained by halving the multiplicities of all strings in T_1, which are assumed to be even;

$T_1 := \emptyset$.

Amplify(T_1, T_2, T_3)

$T_2 := T_1$;

$T_3 := T_1$;
$T_1 := \emptyset$.

Test(T)
If T is empty then "true" else "false".

Init(T, m)
$T := \{i : 1 \leq i \in m\}$.

Note the similarity between Sep and Separate. Of the two, the Sep operation is easier to implement physically and is standard in molecular computation models. Note however that Separate can simulate Sep in polynomial time with no volume overhead. More important, it is well known that Sep can simulate Separate with no time or volume overhead if we use polynomial-size alphabets. In much, if not all, molecular computing, the binary string $b_1 \cdots b_n$ over the alphabet Σ is represented by the molecular sequence $\langle 1, b_1 \rangle \cdots \langle n, b_n \rangle$ over the alphabet $\{1, \ldots, m\} \times \Sigma$ where m is a suitable bound on the length of any string generated during the computation. For convenience, our models will all contain the Separate operation, rather than Sep.

The molecular operation performed at each step is determined by the input string x. Take the 3-SAT problem for example. In the simplest molecular algorithm, the operations to be performed are determined by examining each clause one at a time. The operations corresponding to a particular clause separate out the strands that do not satisfy that clause. We consider only uniform algorithms, in which the sequence of DNA operations to be performed is a polynomial-time computable function (in the classical sense) of the input.

Definition 1.

- Let Q be a set of DNA operations. A Q molecular algorithm is an algorithm that only permits operations in Q.
- The *Time* of a molecular algorithm is the number of operations.
- The *Volume* of a molecular algorithm is the number of DNA sequences.
- A molecular algorithm is $t(n)$-*time uniform* if there is a $t(n)$-time Turing machine T such that for any input, operations can be generated by T.

2.1.3. Molecular Computation Classes

Definition 2. We consider three polynomial-time, $s(n)$-volume bounded and polynomial time uniform models, all of which permit the operations Separate, Pour, Append, and Merge:

- MOL($s(n)$): one tube T_0 is initialized to hold encodings of the numbers $1, \ldots, s(n)$;[1]
- MOL$'$($s(n)$): several tubes are initialized to contain a total of $s(n)$ copies of the empty string, and the Split operation is also permitted;
- MOL-A($s(n)$): T_0 is initialized to contain a single copy of the empty string, the Amplify operation is permitted, and the total number of strings in all tubes must be at most $s(n)$ at all times. All tubes other than T_0 are initialized to be empty. The molecular algorithm *accepts* if T_0 is nonempty after the last molecular operation is performed.

2.1.4. Example: satisfying a simple Boolean formula

This example, finding all satisfying assignments to the formula $(x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$, is from [36]. We will show how to solve this problem in the MOL(), MOL$'$(), and MOL-A() models.

MOL(4) **Algorithm**

MOL(4) Algorithm	T_0	T_1	T_2	T_3	T_4
Initialization	$00, 01, 10, 11$				
Separate($\mathrm{T}_0, 1, 1, \mathrm{T}_1, \mathrm{T}_2$)		$10, 11$	$00, 01$		
Separate($\mathrm{T}_2, 1, 2, \mathrm{T}_2, \mathrm{T}_3$)		$10, 11$	01	00	
Merge($\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_2$)			$10, 11, 01$	00	
Separate($\mathrm{T}_2, 0, 1, \mathrm{T}_1, \mathrm{T}_2$)		01	$10, 11$	00	
Separate($\mathrm{T}_2, 0, 2, \mathrm{T}_2, \mathrm{T}_4$)		01	10	00	11
Merge($\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_0$)	$01, 10$			00	11

[1] One may argue that the initialization step hides Split operations, because that is how this initialization is currently performed in the lab. However, if molecular computing becomes practical, it is reasonable to expect that initial tubes will be prepared outside the computing lab, and that labs will eventually order initial tubes in standard sizes from catalogues, much as they currently order reagents.

MOL$'$(4) **Algorithm**

This is the same as the MOL(4) algorithm, except for the initialization, which must be performed explicitly:

Initialization	T_0	T_1	T_2	T_3	T_4
	$\Lambda, \Lambda, \Lambda, \Lambda$				
$\mathrm{Split}(\mathrm{T}_0, \mathrm{T}_1, \mathrm{T}_2)$		Λ, Λ	Λ, Λ		
$\mathrm{Append}(\mathrm{T}_1, 0)$		$0, 0$	Λ, Λ		
$\mathrm{Append}(\mathrm{T}_2, 1)$		$0, 0$	$1, 1$		
$\mathrm{Merge}(\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_0)$	$0, 0, 1, 1$				
$\mathrm{Split}(\mathrm{T}_0, \mathrm{T}_1, \mathrm{T}_2)$		$0, 1$	$0, 1$		
$\mathrm{Append}(\mathrm{T}_1, 0)$		$00, 10$	$0, 1$		
$\mathrm{Append}(\mathrm{T}_2, 1)$		$00, 10$	$01, 11$		
$\mathrm{Merge}(\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_0)$	$00, 10, 01, 11$				

MOL-A(4) **Algorithm**

This is the same as the MOL(4) algorithm, except for the initialization, which must be performed explicitly:

Initialization	T_0	T_1	T_2	T_3	T_4
	Λ				
$\mathrm{Amplify}(\mathrm{T}_0, \mathrm{T}_1, \mathrm{T}_2)$		Λ	Λ		
$\mathrm{Append}(\mathrm{T}_1, 0)$		0	Λ		
$\mathrm{Append}(\mathrm{T}_2, 1)$		0	1		
$\mathrm{Merge}(\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_0)$	$0, 1$				
$\mathrm{Amplify}(\mathrm{T}_0, \mathrm{T}_1, \mathrm{T}_2)$		$0, 1$	$0, 1$		
$\mathrm{Append}(\mathrm{T}_1, 0)$		$00, 10$	$0, 1$		
$\mathrm{Append}(\mathrm{T}_2, 1)$		$00, 10$	$01, 11$		
$\mathrm{Merge}(\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_0)$	$00, 10, 01, 11$				

2.2. Classical Computation Models

2.2.1. Turing Machines

Turing developed a model for computation, the Turing machine, which gave the theoretical foundation of computation.

Definition 3.

- A Turing machine is a quadruple $M = (K, \Sigma, \delta, s)$. Here K is a finite set of states; $s \in K$ is the initial state. Σ is a finite set of symbols (we say Σ is the *alphabet* of M). We assume K and Σ are disjoint. Σ always contains the special symbols $\sqcup$ and $\triangleright$: The blank and first symbol. Finally δ is a transition function, which maps $K \times \Sigma$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that h (the *halting state*), "yes" (the accepting state), "no" (the *rejecting state*), and the cursor direction $\leftarrow$ for "left," $\rightarrow$ for "right", and $-$ for "stay", are not in $K \cup \Sigma$.

- A Turing machine M is said to terminate in time $f(n)$ in case every computation on an input x of length n consumes time $\leq f(n)$. A deterministic machine M accepts in time $f(n)$ if every terminating computation on an input of length n consumes time $\leq f(n)$. A nondeterministic machine M accepts in time $f(n)$ if for every accepted input of length there is a terminating computation which consumes time $\leq f(n)$.

- $L(M)$ is the language that consists all of strings x that $M(x)$ stops at a specific state called "accept".

- A language L is recognized in time $f(n)$ by machine M if $L = L(M)$ and M terminates in time $f(n)$.

2.2.2. Boolean Formulas and Boolean Circuits

Definition 4.

- A *Boolean formula* is an expression consisting of variables, the operators $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). The precedence is $\neg$ highest, then $\wedge$, then $\vee$. Variables take 0 (false) and 1 (true) values. So do expressions. If E_1 and E_2 are Boolean formula, then $E_1 \wedge E_2$ is 1 if both E_1 and E_2 have value 1, and 0 otherwise. $E_1 \vee E_2$ is 1 if either E_1 or E_2 has value 1, and 0 otherwise. The value of $\neg E_1$ is 1 if E_1 has value 0 and 0 if E_1 has value 1.

- A Boolean formula is *satisfiable* if there is an assignment of 0 and 1's to the variables that gives the expression the value 1.

- A *literal* is either x or $\neg x$ for some variable x.
- A *clause* is $a_1 \vee a_2 \cdots \vee a_k$, where each a_i is a literal. A k*-clause* is a clause with at most k literals.
- A Boolean formula is a *conjunctive normal form* if it is of the form

$$E_1 \wedge E_2 \cdots \wedge \mathrm{E}_k,$$

where each E_i is a clause.

Definition 5.

- *Boolean circuits* are directed acyclic graphs. Some nodes are labeled by variables and called *variable*. The other nodes, called gates, compute simple functions NOT, AND and OR operations. Some of the gates are considered output gates. A boolean circuit naturally computes a function by assigning values to the input nodes, recursively evaluating the gates, and outputting the values of the output gates.
- The complexity of Boolean circuit is measured by its *size*. The size is the number of edges in the circuit.
- Let $\{f_n\}_{n=1}^{\infty}$ be a family of functions. The expression $\mathrm{size}(f_n)$ denotes the size of a minimal circuit that computes f_n. If L is a language over $\{0,1\}^*$, the circuit complexities of L are defined as the circuit complexities of the family $\{L_n\}_{n=1}^{\infty}$, where L_n is characteristic function of the set $L \cap \{0,1\}^n$.

Turing machines and Boolean circuits can simulate each other. Pippenger and Fischer [42] gave an upper bound for a Boolean circuit to simulate a Turing machine.

Theorem 6 (Pippenger and Fischer). *For any language L, $\mathrm{size}(L_n) \leq P \cdot T(n) \cdot \log T(n)$, where $T(n)$ denotes the time complexity of a Turing machine that computes L_n and P denotes its size.*

2.2.3. NP and NP-completeness

Definition 7.

- PF is the class of functions computable in polynomial time by a Turing machine.
- P is the class of languages accepted by polynomial time deterministic Turing machines.
- NP is the class of languages accepted by polynomial time nondeterministic Turing machines.
- A $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction of A to B is a polynomial-time function f such that for each $x \in \Sigma^*$, $x \in A \Longleftrightarrow f(x) \in B$.
- A language B is NP-*complete* if for any A in NP, $A \leq_m^{\mathrm{P}} B$ via some $f \in$ PF.

Many natural problems have been found to be NP-complete. It is widely believed that there is no efficient algorithm in the classical computation model to solve any NP-complete problem. An efficient algorithm is usually defined as an algorithm that runs in a polynomial number of steps. For more information about NP-completeness, please refer to [25]. We list some natural NP-complete problems that we will deal with in this thesis.

SAT Problem Given a formula in conjunctive normal form, is it satisfiable?

3-SAT Problem Given a formula in conjunctive normal form, where each clause is 3-clause, is it satisfiable?

Independent Set Problem Given a graph $G = (V, E)$ and integer k, is there a subset $V' \subseteq V$ containing at least k vertices such that no edge in E connects two vertices of V'.

3-Coloring Problem Given a graph $G = (V, E)$, can all vertices of G be colored by three colors such that no two adjacent vertices have the same color?

Hamiltonian Path Problem Given a directed graph with initial vertex s and target vertex t, is there a directed path that begins at s, ends at t, and enters every other vertex exactly once?

2.3. Bounded Nondeterminism

NP computation with a limited amount of nondeterminism was introduced in [33, 34, 35] and studied further in [16, 20, 41, 15, 26, 50, 29, 10].

Definition 8.

- The class NPbits($b(n)$) consists of languages recognized by an NP machine that makes at most $b(n)$ binary nondeterministic choices on each computation path on inputs of size n. (Actually, prior treatments allowed $O\left(b(n)\right)$ binary choices, but the constant factor turns out to be very important in connection with molecular computation.)
- An NPinit[$s(n)$] Turing machine nondeterministically chooses a number between 1 and $s(n)$, then proceeds deterministically.
- An NPpath[$s(n)$] Turing machine is an NP Turing machine with at most $s(n)$ paths on any length-n input.
- NPinit($s(n)$) is the class of languages accepted by NPinit($s(n)$) machines.
- NPpath[$s(n)$] is the class of languages accepted by NPpath[$s(n)$] machines.

Clearly, NPbits($b(n)$) = NPinit($2^{b(n)}$).

Chapter 3.

Bounded Nondeterminism and Efficient Recursion

3.1. Results for Particular Problems

In the chapter, we show how to transfer recursive algorithms in the classical computation model to a molecular computation model that uses Init, Separate, Merge and Append operations. We improve the molecular algorithms for some NP-complete problems. The results are listed in the following table.

Results	**Previously**			**In This Paper**	
Problem	Space	Limited Model	Reference	Space	Limited Model
Hamiltonian Path	$n!$	$\surd$	[1]		
SAT	2^n	$\surd$	[36]		
QBF	2^n	$\times\times$	[46]		
3-SAT	1.62^n	$\times$	[39]	1.50^n	$\surd$
3-Coloring	1.89^n	$\surd$	[3]	1.35^n	$\surd$
Independent Set	1.51^n	$\surd$	[3]	1.23^n	$\surd$
$(3,2)$-system				1.39^n	$\surd$

(Note: $\times$ means the controversial operation "Amplify" is used and $\times\times$ means the extremely impractical operation "Intersection" is used in algorithms respectively.)

3.2. $\mathrm{NPinit}(s(n)) \subseteq \mathrm{MOL}(s(n))$

In this section we show how to simulate bounded nondeterministic computation via bounded-volume molecular computation. Results of this type appear in [6, 46, 47, 55], but they assume models of molecular computation with more powerful operations, such as Amplify, that may be harder to implement in practice. Independently, Boneh et al. [13] obtained a result similar to ours.

Lemma 9. *Let π be a circuit with m gates. Given a tube T_0, a molecular algorithm using only the operations Pour, Append, and Merge, running in time $O(m)$, and using only four test tubes can create tubes T_1 and T_2 such that T_1 contains all strings z from tube T_0 that satisfy $\pi(z) = 1$ and T_2 contains all strings z from tube T_0 that satisfy $\pi(z) = 0$.*

Proof: Let π's input gates be $g_1, \ldots, g_n$ and internal gates be $g_{n+1}, \ldots, g_m$ in topological order; in particular g_m is the output gate. We will use four tubes T_0, T_1, T_2, T_3. For each i, let g_i compute $f_i(g_{j(i)}, g_{k(i)})$ where $j(i) < i$, $k(i) < i$, and f_i is a binary function. We perform the following algorithm:

(1) for $i := n+1$ to m do
(2) Separate($T_0, 0, j(i), T_1, T_2$)
(3) Separate($T_1, 0, k(i), T_0, T_3$)
(4) Append($T_0, f_i(0,0)$)
(5) Append($T_3, f_i(0,1)$)
(6) Merge(T_0, T_3, T_1)
(7) Separate($T_2, 0, k(i), T_0, T_3$)
(8) Append($T_0, f_i(1,0)$)
(9) Append($T_3, f_i(1,1)$)
(10) Merge(T_0, T_3, T_2)
(11) Merge(T_1, T_2, T_0)
(12) Separate($T_0, 0, m, T_1, T_2$)

Throughout the computation, value of gate g_i is stored at ith bit of each sequence for $i = 1, \ldots, m$. Let's check each cycle of the loop for $n+1 \leq i \leq m$. Right after (4), T_0 contains all sequences whose $j(i)$th bit and $k(i)$th bit are $0, 0$; and T_3 contains all sequences whose $j(i)$th bit and $k(i)$th bit are $0, 1$ respectively. In (4), we append $f_i(0,0)$ to all sequences in T_0 as all of them have the same value $f_i(0,0)$ at gate g_i. Similarly we append $f_i(0,1)$ to all sequences in T_3. Right after (7), T_0 contains all sequences whose $j(i)$th bit and $k(i)$th bit are $1, 0$ respectively, and T_3 contains all sequences whose $j(i)$th bit and $k(i)$th bit are $1, 1$. At (8), we append $f_i(1,0)$ to all sequences in T_0 as all of them have the same value $f_i(1,0)$ at gate g_i. Similarly we append $f_i(1,1)$ to all sequences in T_3 at (9). At completion, T_1 contains all strings that satisfy π and T_2 contains all strings that do not satisfy π. ∎

Theorem 10. NPinit$(s(n)) \subseteq$ MOL$(s(n))$.

Proof: Let L be accepted by an NPinit$(s(n))$ machine M. Construct a deterministic machine M' that takes as inputs a string x and a positive integer $z \leq s(n)$ and accepts iff M accepts input x with nondeterministic guess z. Obtain M'_x by fixing the input x, so the only input to M'_x is the number z. Construct a circuit π equivalent to M'_x in the usual way (see [42]). Apply Lemma 9 to π to see that L is in MOL$(s(n))$. ∎

3.3. Enumerating Search Spaces with Bounded Nondeterminism

Let's start with an important example that illustrates the general technique. How do you generate all strings consisting of k 1's and $n-k$ 0's? Bach et al. [3] show how to do so efficiently on a molecular computer using generalized Split() operations. We will show how to do so using bounded nondeterminism; consequently, generalized Split() operations are not necessary for this problem.

We start with an ordinary recursive algorithm that prints all strings of the desired form in lexicographical order. Then we convert that algorithm into a function that maps the number i to the ith string having the desired form. Since the numbers $1, \ldots, \binom{n}{k}$ can be enumerated with $\lceil \log \binom{n}{k} \rceil$ nondeterministic bits, so can the corresponding strings.

```
procedure Gen(x, n, k)
if n = 0 then
    print x
else
    if k < n then
        Gen(x0, n − 1, k)
```

if $k > 0$ then

Gen$(x1, n-1, k-1)$

A call to Gen(Λ, n, k) generates all strings consisting of k 1's and $n-k$ 0's, in lexicographical order. We determine the ith string output by Gen as follows:

procedure Enum(i, x, n, k)

if $n = 0$ then

print x

else

if $i \leq \binom{n-1}{k}$ then

Enum$(i, x0, n-1, k)$

else

Enum$(i - \binom{n-1}{k}, x1, n-1, k-1)$

A call to Enum(i, Λ, n, k) produces the ith string (in lexicographical order) consisting of k 1's and $n-k$ 0's.

Why were we able to convert the recursive algorithm to a bounded-nondeterministic one? Because we were able to compute in polynomial time the number of strings generated by any recursive call. When a recurrence depends only on a fixed number of polynomial-bounded parameters (in this case n and k, but not x), we can solve it in polynomial time via dynamic programming. If the recurrence depends on x (which is typical), then we compute a bound that is independent of x (in practice, such a bound is usually given along with the recursive algorithm); that bound can be computed in polynomial time by dynamic programming.

Note that our technique applies only to recursive algorithms whose running time bound is obtained simply by summing the running time bounds for its recursive calls. In particular our technique does not apply to table-based recursive algorithms. Nor does our technique apply to more general self-reductions, such as those used for QBF.

3.3.1. Implementing Recursion with Bounded Nondeterminism

Recursive algorithms for NP problems usually take the form of d-self-reductions ("d" for disjunctive). Self-reductions were defined in [52] and d-self-reductions were defined in [53].

Definition 11. Let $|y|$ denote the size of the problem instance y. A partial order $\prec$ is polynomial well-founded if there exists a polynomial-bounded function p such that

- $y_m \prec \cdots \prec y_1 \Rightarrow m \leq p(|y_1|)$
- $y_m \prec \cdots \prec y_1 \Rightarrow |y_m| \leq p(|y_1|)$

For technical simplicity we will consider only languages L such that $\Lambda \in L$.

Definition 12. A d-self-reduction for a language L consists of a polynomial time computable function $h(x) = \{x_1, \ldots, x_m\}$ and a polynomial-well-founded partial order $\prec$ on problem instances such that

- Λ is the only minimal element under $\prec$
- for all $x \neq \Lambda$, $x \in L \iff h(x) \cap L \neq \emptyset$
- for all x, $x_i \in h(x) \Rightarrow x_i \prec x$

Definition 13. Let $\langle h, \prec \rangle$ be a d-self-reduction and let x be a problem instance.

- $T_{h,\prec}(x)$ is the tree rooted at x in which each node y's children are the elements of $h(y)$.
- $|T_{h,\prec}(x)|$ is the number of leaves in $T_{h,\prec}(x)$.

If $\langle h, \prec \rangle$ is a self-reduction for L, then the corresponding recursive algorithm for L runs in time $|x|^{O(1)} |T_{h,\prec}(x)|$. The analysis of such an algorithm usually provides a bound on $|T_{h,\prec}(x)|$ that is suitable for use in constructing a molecular algorithm for L. We formalize this below:

Definition 14. Let T be a polynomial-time computable function. A language L is in REC$(T(x))$ if there is a d-self-reduction $\langle h, \prec \rangle$ for L such that for all x

(1) $|T_{h,\prec}(x)| \leq T(x)$, and

(2) $T(x) \geq \sum_{x_i \in h(x)} T(x_i)$.

Lest conditions (1) and (2) above seem restrictive, we argue that they are quite natural. We consider the typical analysis of a recursive algorithm. One introduces a function T and proves by induction on $|x|$ that $|T_{h,\prec}(x)| \leq T(x)$, which is (1). The inductive hypothesis is that $|T_{h,\prec}(w)| \leq T(w)$ if $|w| < |x|$. Inspection of the algorithm yields

$$\begin{aligned} |T_{h,\prec}(x)| &= \sum_{x_i \in h(x)} T_{h,\prec}(x_i) \\ &\leq \sum_{x_i \in h(x)} T(x_i) \qquad \text{by the inductive hypothesis} \end{aligned}$$

The last step in the induction consists of showing that T satisfies $\sum_{x_i \in h(x)} T(x_i) \leq T(x)$, which is (2). The only other requirement on T is that T be polynomial-time computable. We will deal with that later in this section.

The function T above depends on problem instances rather than their size because the analysis of the algorithm may depend on two or more parameters. We will need an analogous variant of NPinit().

Definition 15. $\text{NPinit}'(S(x))$ consists of languages recognized by NP machines that nondeterministically choose a number between 1 and $S(x)$ on input x and then behave deterministically.

Clearly, if $S(x) \leq s(|x|)$ then $\text{NPinit}'(S(x)) \subseteq \text{NPinit}(s(n))$.

Theorem 16. $\text{REC}(T(x)) \subseteq \text{NPinit}'(T(x))$.

Proof: Let $L \in \text{REC}(T(x))$ via $\langle h, \prec \rangle$. We will define a deterministic polynomial-time computable function $\text{path}(i, x)$ taking values in $\{0, 1, \Lambda\}$ such that $\text{path}(1, x) \cdots \text{path}(T(x), x)$ is equal to the sequence of values at the leaves of $T_{h,\prec}(x)$ in canonical order. The proof is completed by having the ith path of an $\text{NPinit}'(T(x))$ machine compute $\text{path}(i, x)$; clearly that machine accepts L. The function $\text{path}(i, x)$ will be computed via tail recursion.

function $\text{path}(i, x)$

if $x = \Lambda$ then

```
        return true
    else if h(x) = ∅ then
        return false
    else
        {x_1, ..., x_m} := h(x)
        for j := 1 to m do
            if i ≤ T(x_j) then
                return path(i, x_j)
            else
                i := i − T(x_j)
        return Λ
```

∎

Now we give sufficient conditions for T to be polynomial-time computable.

Definition 17. We say that a partial order $\prec$ on problem instances is *parameterizable* if there is a function m from problem instances to a set M, a partial order $\prec'$ on M, and a polynomial p such that

- $m(x)$ is computable in time polynomial in $|x|$, and
- $x \prec y \Rightarrow m(x) \prec' m(y)$, and
- $||\{i : i \prec' m(x)\}|| \leq p(|x|)$.

In many examples we will take $m(x) = |x|$ and $\prec'$ to be the standard linear order on natural numbers. In other examples, $m(x)$ will be a tuple of parameters (such as the number of 2-clauses and the number of 3-clauses in a Boolean formula); in many (but not all) of these examples we use the partial order $(a_1, \ldots, a_k) \prec' (b_1, \ldots, b_k)$ if $(\forall i)[a_i \leq b_i]$ and $(\exists i)[a_i < b_i]$.

Definition 18. Given h and m, define

- $\mathrm{mh}(x) =$ the multi-set $\{m(x_i) : x_i \in h(x)\}$
- $\mathrm{MH}(x) =$ the set $\{\mathrm{mh}(y) : m(y) = m(x)\}$

Definition 19. A d-self-reduction $\langle h, \prec \rangle$ is *by cases* if $\prec$ is parameterizable via $\langle m, \prec' \rangle$ in such a way that $\mathrm{MH}(x)$ is computable in time polynomial in $|x|$.

Lemma 20. *Let $\langle h, \prec \rangle$ be a d-self-reduction by cases with parameter function $m()$. Let T_0 be the least function T such that*

(1) $|T_{h,\prec}(x)| \leq T(x)$

(2) $T(x) \geq \sum_{x_i \in h(x)} T(x_i)$

(3) $T(x)$ *is a function of* $m(x)$

Then T_0 exists and $T_0(x)$ is computable in time polynomial in $|x|$.

Proof: Let $\langle h, \prec \rangle$ have a parameterization $\langle m, \prec' \rangle$, where $m(x)$ and $\mathrm{MH}(x)$ are computable in time polynomial in $|x|$. Define a partial function t from M to natural numbers recursively:

$$t(\mu) = \begin{cases} 1 & \text{if } \mu \text{ is a minimal element under } \prec' \\ \max_{m(y)=\mu} \sum_{y_i \in h(y)} t(m(y_i)) & \text{otherwise} \end{cases}$$

If μ is in the range of m, then $t(\mu)$ is defined because $\mathrm{MH}(x)$ is a finite set for every x. Now it is easy to see that $t \circ m$ is the least function satisfying (1,2,3).

By Definition 17, $|\{i : i \prec' m(x)\}| \leq p(|x|)$. If we compute $t(m(x))$ by the obvious recursion, at most $p(|x|)$ different subproblems will arise. If we use a table to avoid recomputation, the recursion will run in polynomial time. ∎

3.3.2. 3-SAT

In this section, we apply our results to the classic 3-SAT algorithm of Monien and Speckenmeyer [37] and a recent unverified 3-SAT algorithm of Schiermeyer [51]. The former yields a simple $\mathrm{MOL}(1.62^n)$ algorithm, and the latter (assuming that Schiermeyer's paper is correct), yields a $\mathrm{MOL}(1.497^n)$ algorithm.

Monien and Speckenmeyer's Algorithm

The size of a satisfiability instance is the number of variables. Let $f|_\ell$ denote the formulas obtained by replacing in f the literal ℓ by true and $\overline{\ell}$ by false. A k-clause is a disjunction of k literals. The function 3SAT takes a formula f consisting of some 3-clauses and at least one 1-clause or 2-clause.

> function 3SAT(f)
> if f is the empty set of clauses then
>   return true
> else if f contains an empty clause then
>   return false
> else if some variable v appears only in positive literals then
>   return 3SAT($f|_v$)
> else if some variable v appears only in negative literals then
>   return 3SAT($f|_{\overline{v}}$)
> else if f contains a clause C consisting of a single literal ℓ then
>   return 3SAT($f|_\ell$)
> else if f contains a clause C consisting of two literals ℓ_1, ℓ_2 then
>   return 3SAT($f|_{\ell_1}$) $\vee$ 3SAT($f|_{\overline{\ell_1}}|_{\ell_2}$)
> else
>   let v be the first variable to appear in f
>   return 3SAT($f|_v$) $\vee$ 3SAT($f|_{\overline{v}}$)

The last case in the recursion is ostensibly the worst, yielding two subproblems of size $n-1$, but it only occurs on the first call or immediately after eliminating a single variable, which yields a single subproblem of size $n-1$; unrolling the recursion, we see that the last case gives two subproblems of size $n-2$. The worst case is the second to the last, which yields subproblems of size $n-1$ and $n-2$. Thus the number of leaves in the self-reduction is at most $2f(n)$ where $f(n)$ is given by the recurrence $f(n) = f(n-1) + f(n-2)$; in particular $2f(n) \leq 1.62^n$ for almost all n.

The algorithm above is clearly a d-self-reduction for 3-SAT. The value function h for a formula is the set of subformulas generated by the recursive algorithm. Let $m(x) = n$, where n is the number of variables in the formula x. $\leq$ is the normal order for the integers. From the analysis above we know $\mathrm{mh}(x)$ is either $\{n-1\}, \{n-2, n-2\}$ or $\{n-2, n-1\}$. $\mathrm{MH}(x)$ is $\{\{n-1\}, \{n-2, n-2\}, \{n-2, n-1\}\}$ that is clearly polynomial time computable. Let $t(n) = 1.62^n$. $2f(n) \leq t(n)$. Hence $t(n)$ is an upper bound of the number of leaves of computation tree for the recursive algorithm. It is easy to see that $t(n) \geq t(n-2)+t(n-1) \geq t(n-2)+t(n-2)$. Hence, $T_{3\mathrm{S}}(x) = t(m(x))$ satisfies the conditions of Lemma 20 and 3-SAT is in $\mathrm{REC}(T_0(F))$ for some $T_0 \leq T_{3\mathrm{S}}$. By Theorem 16 and Theorem 10, $3\mathrm{SAT} \in \mathrm{NPinit}(t_{3\mathrm{S}}(n))$, so 3-SAT is in $\mathrm{MOL}(1.62^n)$. The same volume bound for 3-SAT was obtained previously by Ogihara [39], but in a model that allows more powerful operations like Polymerization, which can implement the Amplify operation.

Schiermeyer's Algorithm

Schiermeyer [51] reports a 1.497^n time algorithm for the 3-SAT problem. His algorithm is a d-self-reduction for the 3-SAT problem. We will prove that $3\mathrm{SAT} \in \mathrm{REC}(T(F))$, where the function $T(F) \leq 1.497^n$ and will be defined below. We follow [51] to define F_3 and F_3'. For a formula F with n variables, let p be the maximum number of 1-clauses and 2-clauses (with preference of 1-clauses) such that no variable occurs more than twice. Let q be the number of remaining 2-clauses and define $m = p + \min(2, q)$. Let $F_3(n) = c\beta^n \cdot \frac{\beta}{\alpha^4}$ and $F_3'(n) = c\beta\alpha^{-m}$, where $\beta = 1.4963, \alpha = 1.04855$ and c is a sufficiently large constant.

$$T(F) = \begin{cases} F_3(n) & \text{if } F \text{ has no 1-clauses or 2-clauses} \\ F_3'(n) & \text{otherwise} \end{cases}$$

Schiermeyer states that $F_3'(n) \geq |T_{h,\prec}(F)|$ if F has at least one 1-clause or 2 clause, and that $F_3(n) \geq |T_{h,\prec}(F)|$ for all F. Hence, $T(F) \geq |T_{h,\prec}|$, since $|T_{h,\prec}| \leq$ the number of recursive calls. The inequalities that Schiermeyer gives in the proofs of his Lemma 4.3 and Lemma 4.4 imply that our $T(F)$ satisfies the conditions of Definition 14. Hence, $3\mathrm{SAT} \in \mathrm{REC}(T(F)) \subseteq \mathrm{MOL}(1.497^n)$.

3.3.3. 3-Coloring and the $(3, 2)$-System Problem

Beigel and Eppstein [8] give algorithms for the $(3, 2)$-system problem and 3-coloring. In the (a, b)-system problem, we are given a collection of n vertices, each of which can be given one of a different colors. However certain color combinations are disallowed: we are also given a set of constraints, each of which forbids one coloring of some b-tuple of variables. The $(3,2)$-system problem generalizes 3-coloring, 3-SAT and 3-edge-coloring.

A $(3,2)$-System Algorithm

The size of a $(3,2)$ system is the number of variables in it. Beigel and Eppstein's [8] $(3,2)$-system algorithm can be sketched as follows:

```
function 32SYS(F)
if |F| ≤ 5 then
    return brute-force(F)
else
    ⟨F_1, ..., F_k⟩ = h_32(F)
    return ⋁_{i=1}^{m} 32SYS(F_i)
```

In the algorithm above, brute-force(F) means "use the brute force method to solve the $(3,2)$-system F;" $k \leq 3$; h_{32} is polynomial-time computable; and $|F_i| < |F|$. Let $h = h_{32}$ and let $\prec$ be the standard linear ordering on the natural numbers. Then $\langle h, \prec \rangle$ is a d-self-reduction for $(3,2)$-system. Define $m(F) = |F|$.

In case 1, $\mathrm{mh}(F) = \{n-(4+i), n-1\}$, where $i \geq 0$.

In cases 2a, 2c, and 3, $\mathrm{mh}(F) = \{n-(3+i), n-2\}$, where $i \geq 0$.

In cases 2b, 2d, 6, 8c and 9, $\mathrm{mh}(F) = \{n'\}$, where $n' < n$.

In case 4, $\mathrm{mh}(F) = \{n-5, n-3, n-3\}$.

In case 5, $\mathrm{mh}(F) = \{n-4, n-4\}$.

In case 7, $\mathrm{mh}(F) = \{n-3, n-3\}$.

$\mathrm{MH}(F)$ is polynomial-time computable by the case analysis above. Let $t(n) = 1.38028^n$. It is easy to see that for every input x with $\{n_1, \ldots, n_k\} = \mathrm{mh}(x)$ and $m(F) = n$, $t(n) \geq t(n_1) + \cdots + t(n_k)$. Define $T(F) = t(m(F))$. By Lemma 20, there is a polynomial-time computable function T_0 such that $T_0(F) \leq T(F)$ and T_0, h, and $\prec$ satisfy the conditions of Definition 14. Thus, $(3,2)$-system is in $\mathrm{REC}(T_0(F))$. So,

$$
\begin{aligned}
(3,2)\text{-system} &\in \mathrm{NPinit}'(T_0(F)) && \text{by Theorem 16} \\
&\subseteq \mathrm{NPinit}'(T(F)) && \text{because } T_0(F) \leq T(F)
\end{aligned}
$$

$$
\begin{aligned}
&= \text{NPinit}(t(n)) \qquad \text{because } T(F) = t(|F|) \\
&= \text{NPinit}(1.38028^n) \qquad \text{because } t(n) = 1.38028^n \\
&\subseteq \text{MOL}(1.38028^n) \qquad \text{by Theorem 10.}
\end{aligned}
$$

3-Coloring Algorithm

There are two parts to Beigel and Eppstein's algorithm. The first part runs in polynomial time and finds an independent set S with a lot of neighbors. Let $\Gamma(S)$ denote the set of vertices in G that are not in S but are adjacent to an element of S. The second part 3-colors S in all possible ways. Each of these $3^{|S|}$ partially-colored graphs is transformed into an equivalent $(3,2)$-system with $n - |S| - |\Gamma(S)|$ variables, which is solved by calling 32SYS. Their algorithm runs in time $3^{|S|}1.38028^{n-|S|-|\Gamma(S)|}$, which is less than 1.345^n for sufficiently large n. Thus we have the following NPinit(1.345^n) algorithm:

choose a natural number $m < 1.345^n$
construct Beigel and Eppstein's set S
let $c = m \bmod 3^{|S|}$
color S with the cth 3-coloring in the lexicographical ordering
form the corresponding $(3,2)$-system F
let $b = \lfloor m/3^{|S|} \rfloor$
run 32SYS(F) using the nondeterministic choices dictated by b

Therefore 3-coloring is in MOL(1.345^n).

3.3.4. Independent Set

For a graph G, an independent set S is a subset of G's nodes such that there is no edge between any two nodes in S. The independent set problem is "given a graph G and a number k, does G contain an independent set of cardinality at lest k?"

Tarjan's Algorithm

Consider the following simple algorithm due to Tarjan [56]. (In the description below, $d(v)$ denotes the degree of v, and $N(v)$ denotes the neighbor set of v. The expression

$\max(S, T)$ denotes the larger of the two sets S and T, with ties resolved arbitrarily.)

function MIS(G)
pick any vertex v in G
if $d(v) \leq 1$ then
    return $\{v\} \cup \text{MIS}(G - v - N(v))$
else
    return $\max(\text{MIS}(G - v), \{v\} \cup \text{MIS}(G - v - N(v)))$

This is a self-reduction with at most $T(n)$ leaves where $T(n)$ satisfies $T(n) = T(n-1) + T(n-3)$ where $T(n)$. The recurrence can be solved in polynomial time by an explicit formula or by dynamic programming so the independent set problem is in $\text{MOL}(1.47^n)$, which is better than prior results [3]. Because the algorithm is particularly simple, the molecular algorithm can even be made to run in linear time.

Robson's Algorithm

The best published purely recursive algorithm for the independent set problem is due to Robson [45] and runs in time 1.229^n for sufficiently large n. A d-self-reduction with 1.229^n leaves is evident from Robson's paper, so we have we have a $\text{MOL}(1.229^n)$ algorithm for the independent set problem. Details are given in the Appendix.

Robson has a faster dynamic programming algorithm for independent set, but we see no way to adapt it to molecular computing. Molecular computing may motivate the search for efficient recursive algorithms that do not use dynamic programming. Towards that end we have found a recursive 1.223^n time (for sufficiently large n) algorithm for independent set [7] that is based on a d-self-reduction and hence is directly adaptable to molecular computing.

Let us consider the following two related problems. MAX-INDEPENDENT problem: given a graph G, what is the size of the largest independent set of G? INDEPENDENT problem: given a graph G and a number k, is there an independent set of G with size at least k? The first problem is P^{NP}-complete and the second problem is NP-complete. For these problems, n denotes the number of vertices in G.

Robson's algorithm deals with the following input and output. The input (G, S, a) has the following properties: (1) $a \in \{0, 1, 2\}$; (2) if $a = 1$ then $|S| = 2$, and (3) if

$a = 0$ then $S = \emptyset$. The output is the size of the largest independent set I containing at least a elements of S. Robson's algorithm can be expressed as the following form:

function MAXIND(F)
$\langle G, S, a\rangle = F$
if $|G| \leq 5$ then
return brute-force(F)
else
$\langle F_1, \ldots, F_m\rangle = h_1(F)$
$\langle b_1, \ldots, b_m\rangle = g_1(F)$
return max($b_1 +$ MAXIND(F_1), ..., $b_k +$ MAXIND(F_m))

In the above algorithm h_1 produces a series of new problems and g produces a tuple of integers. Without loss of generality, assume that the graph in F_i has fewer nodes than that in F (otherwise, we unroll the recursion as we did for 3SAT).

From Robson's algorithm, both h_1 and g_1 are polynomial time computable. MAXIND$(G, \emptyset, 0)$ returns the size of G's largest independent set. Define

$$\langle G, S, a\rangle \prec_1 \langle G', S', a'\rangle \iff |G| < |G'|.$$

Let $T'_{h_1,\prec_1}(F)$ denote the computation tree of the algorithm MAXIND on input F.

Define $T'(F)$ as follows (t, t^1 and t^2 are defined according to Robson's paper):

$$T'(F) = \begin{cases} t(G, n) & \text{if } F = \langle G, S, 0\rangle \\ t^1(G, |G|, S, |S|) & \text{if } F = \langle G, S, 1\rangle \\ t^2(G, |G|, S, |S|) & \text{if } F = \langle G, S, 2\rangle \end{cases}$$

From the proof of Theorem 1 and the algorithms for ms_1 and ms_2 in [45], we know that

$$T'(F) \geq \max\Big(|T'_{h_1,\prec}(F)|, T'(F_1) + \cdots + T'(F_m)\Big).$$

It is easy to transform the algorithm MAXIND into a d-self-reduction for a generalization of the independent set problem.

The input will be $\langle\langle G, S, a\rangle, k\rangle$ where

- $a \in \{0, 1, 2\}$,
- $a = 0 \Rightarrow S = \emptyset$,

- $a = 1 \Rightarrow |S| = 2$, and

The output is true iff G contains a maximal independent set consisting of at least k nodes, at least a of which belong to S.

function IND(F, k)
$\langle G, S, a\rangle = F$
if $|G| \leq 5$ then
    return brute-force(F)
else
    $\langle F_1, \ldots, F_m\rangle = h_1(F)$
    $\langle b_1, \ldots, b_m\rangle = g_1(F)$
    if $k \leq b_i$ then
        return true
    else
        return $\bigvee_{i=1}^{m}$ IND$(F_i, k - b_i)$

Define

$$\langle\langle G, S, a\rangle, k\rangle \prec \langle\langle G', S', a'\rangle, k'\rangle \iff \langle G, S, a\rangle \prec_1 \langle G', S', a'\rangle.$$

If $k > b_i$ for $i = 1, \ldots, m$, then let

$$\begin{aligned}
\langle F_1, \ldots, F_m\rangle &= h_1(F), \\
\langle b_1, \ldots, b_m\rangle &= g_1(F), \\
h(F, k) &= \langle\langle F_1, k - b_1\rangle, \ldots, \langle F_m, k - b_m\rangle\rangle.
\end{aligned}$$

If $k \leq b_i$ for some i, the recursive algorithm returns "true" and h is undefined.

The above algorithm ($\langle h, \prec\rangle$) is a d-self-reduction for INDEPENDENT. Clearly (G, k) is in the problem INDEPENDENT if and only if IND$(\langle G, \emptyset, 0\rangle, k) =$ true. It is easy to see that

$$T'_{h_1, \prec_1}(F) \geq T_{h, \prec}(F, k).$$

Define $T(F, k) = T'(F)$. We have

$$T(F, k) = T'(F) \geq |T_{h_1, \prec_1}| \geq |T_{h, \prec}|.$$

Because $T'(F) \geq T'(F_1) + \cdots + T'(F_m)$ for $\langle F_1, \ldots, F_m \rangle = h(F)$, we have

$$T(F, k) \geq T(F_1, k - b_1) + \cdots + T(F_m, k - b_m).$$

Hence INDEPENDENT belongs to $\mathrm{REC}(T'(F,k)) \subseteq \mathrm{NPinit}(T'(F,k))$, so the independent set problem is in $\mathrm{MOL}(1.229^n)$.

3.4. Computation Model Based of Surface Chemistry

Cai, Condon, et al.,[17] introduced a new model of DNA computation based on surface chemistry and proved it can simulate circuits. Similar results as Lemma 9 and Theorem 10 also hold in their model. The recursive algorithms implemented in this paper can also be implemented in their model.

Chapter 4.

A Comparison of Resource-Bounded Molecular Computation Models

In this chapter we determine the relations among several models of molecular computation and Turing machine computation. We consider three polynomial-time, $s(n)$-volume bounded models, $\mathrm{MOL}(s(n))$, $\mathrm{MOL}'(s(n))$, and $\text{MOL-A}(s(n))$ model. We consider two polynomial-time, $s(n)$-path bounded models of nondeterministic computation: $\mathrm{NPinit}(s(n))$ and $\mathrm{NPpaths}(s(n))$ We prove

- $\mathrm{MOL}'(s(n)) = \mathrm{MOL}(s(n)) = \mathrm{NPinit}(s(n))$
- $\text{MOL-A}(s(n)) = \mathrm{NPpaths}(s(n))$

Thus the Split operation does not increase the power of polynomial-time molecular computation. We conjecture that $\mathrm{NPinit}(s(n)) \subset \mathrm{NPpaths}(s(n))$, so we believe that the Amplify operation does increase the power of polynomial-time molecular computation.

Finally, we note that Winfree [61] has proved some relations between branching programs and memoryless molecular computation (where the Append operation is not allowed). His results suggest that the Amplify operation might increase the power of molecular computation. However, he considers only time complexity and not volume complexity for molecular computation. Furthermore, only brute-force algorithms have been implemented in the memoryless models he describes.

4.1. $\mathrm{MOL}'(s(n)) = \mathrm{MOL}(s(n)) = \mathrm{NPinit}(s(n))$

First, it is easy to see that $\mathrm{MOL}(s(n)) \subseteq \mathrm{MOL}'(s(n))$. Write $s(n) = 2^{a_1} + \cdots + 2^{a_k}$ where $a_1 > \cdots > a_k$, and initialize tube T_i to contain 2^{a_i} copies of the empty string for $i = 1, \ldots, k$. Then use Split, Append, and Merge to generate the numbers $2^{a_1} + \cdots + 2^{a_{i-1}} + 1, \ldots, 2^{a_1} + \cdots + 2^{a_i}$ in tube T_i. Second, By Theorem 10, we know that $\mathrm{NPinit}(s(n)) \subseteq \mathrm{MOL}(s(n))$. In this section we will show that $\mathrm{MOL}'(s(n)) \subseteq \mathrm{NPinit}(s(n))$. Thus $\mathrm{MOL}'(s(n)) = \mathrm{MOL}(s(n)) = \mathrm{NPinit}(s(n))$.

Theorem 21. $\mathrm{MOL}'(s(n)) \subseteq \mathrm{NPinit}(s(n))$

Proof: We think of the empty strings initially placed in T_0 by the $\mathrm{MOL}'(s(n))$ computation, although identical, as being numbered (or named) arbitrarily 1 through $s(n)$. Without loss of generality, we may assume that the Split operation partitions identical strings by putting the lower numbered strings in the lower numbered destination tube.

The kth computation of the $\mathrm{NPinit}(s(n))$ machine will be responsible for the kth string in the $\mathrm{MOL}'(s(n))$ computation. If the kth computation is responsible for a string z at time t, it will determine the location of all prefixes of z in the $\mathrm{MOL}'(s(n))$ computation (including duplicates) at time t, so in particular it determines the location of the string it is responsible for. The collection of strings identical to p in a tube T is represented as a finite union of intervals.

No operation besides Split increases the number of intervals for prefixes of the string z. (Append can increase the number of intervals for one prefix, but only at the expense of other prefixes.) For each string p the number of intervals consisting of copies of p increases by at most 1 per Split operation. Therefore the number of intervals consisting of copies of some prefix of z at time t is at most $(|z|+1)(t+1) \leq t(t+1) = O(t^2)$. Thus this collection of intervals may be represented in polynomial space. It remains to show how to compute it in polynomial time.

The NP algorithm is divided into stages, 1 for each step of the molecular computation. At stage t, the ith path determines what string z it is responsible for at time t, based on what string it was responsible for at time $t-1$, that string's location at time $t-1$, and the operation performed at time $t-1$. (The string changes only if an Append is performed at time t.) Then the ith path simulates the molecular computation for steps 1 through t, determining the locations of all prefixes of z.

Now we describe how to simulate the molecular computation. At time 0, T_0 contains copies 1 through $s(n)$ of the empty string, and all other tubes are empty. It remains to show how simulate the tth step in polynomial time.

Whatever operation is performed at time t, the prefixes of z in tubes not addressed by that operation are the same as at time $t-1$. We take several cases, depending on what kind of operation is performed at time t:

Separate(T_1, c, i, T_2, T_3)**.** T_1 contains no prefix of z. Let p be any prefix of z. If the ith character of p is c then T_2 contains the copies of p that were in T_1 at time $t-1$ and T_3 contains no copies of p; otherwise T_3 contains the copies of p that were in T_1 at time $t-1$ and T_2 contains no copies of p. If p is of length $< i$, then p is contained in T_3.

Pour(T_1, T_2) T_1 contains no prefix of z. Let p be any prefix of z. T_2 contains the copies of p that were in T_1 at time $t-1$.

Append(T, c) Let pc be any prefix of z that ends with c. T contains the copies of pc corresponding to the copies of p that it contained at time $t-1$.

Merge(T_1, T_2, T_3) T_1 and T_2 contain no prefixes of z. T_3 contains the prefixes of z that were in T_1 or T_2 at time $t-1$.

Split(T_1, T_2, T_3) T_1 contains no prefix of z. For each prefix p of z, T_2 contains the lower-numbered half of the copies of p that were in T_1 at time $t-1$ and T_3 contains the remaining copies. ∎

Corollary 22. $\mathrm{MOL}'(s(n)) = \mathrm{MOL}(s(n)) = \mathrm{NPinit}(s(n))$

Note 1: Bach et al [3] define an asymmetric split operation that splits T_1 in a specified ratio $r : 1-r$. An ordinary split cannot simulate even an asymmetric split because an ordinary split is undefined on a tube with an odd number of elements.

However, our proof of Theorem 21 works for asymmetric splits (just replace one half by the fraction r). Thus asymmetric splits are no better than ordinary splits as far as volume is concerned.

Note 2: In our simulation and in [9, 13] there is a 1–1 correspondence between molecular sequences and computational paths. Thus the counting versions of the classes above also coincide.

4.2. MOL-A$(s(n))$ = NPpaths$(s(n))$

Lemma 23. MOL-A$(s(n)) \subseteq$ NPpaths$(s(n))$

Proof: Each path of the NPpaths$(s(n))$ machine will be responsible a molecular sequence z, *i.e.*, the path knows which test tube z is in. At time 0, the NPpaths$(s(n))$ machine has just one path, which is responsible for Λ, which is in T_0.

To simulate a single step of the MOL-A$(s(n))$ computation, we take several cases, depending on what kind of operation is performed at time t:

Separate(T_1, c, i, T_2, T_3)**.** If z was in T_1 at time $t-1$ and the ith character of z is c, then z is in T_2 at time t. If z was in T_1 at time $t-1$ and the ith character of z is not c, then z is in T_3 at time t.

Pour(T_1, T_2) If z was in T_1 at time $t-1$ then z is in T_2 at time t.

Append(T, c) If z was in T at time $t-1$, then sc is in T at time t.

Merge(T_1, T_2, T_3) If z was in T_1 or T_2 at time $t-1$, then z is in T_3 at time t.

Amplify(T_1, T_2, T_3) If z is in T_1 at time $t-1$, then z is in both T_2 and T_3 at time t. ▮

Lemma 24. NPpaths$(s(n)) \subseteq$ MOL-A$(s(n))$

Proof: Without loss of generality, assume the NPpaths($s(n)$) machine makes only binary choices. Call its input x. Modify the machine so that it behaves deterministically — all choices specified by an auxiliary binary input y. Convert that machine to an equivalent circuit by the techniques of [42]. For a given x, we wish to evaluate that circuit for all $s(|x|)$ possible y using molecular computation. The molecular algorithm evaluates the circuit bottom up. If a gate depends on the jth bit of y then the algorithm first separates sequences in which the value of this bit is not already stored. It amplifies those sequences, appends $\langle j, 0\rangle$ to half of them and $\langle j, 1\rangle$ to the other half, merges the two halves, and then merges in the sequences in which the bit was already stored. If the gate depends on two bits of y, then this procedure is repeated for the second bit. At this point the gate's value is determined, and can be computed using the techniques of [9, 13]. ∎

Thus we have

Theorem 25. NPpaths($s(n)$) = MOL-A($s(n)$)

Note that this section's simulations provide a 1–1 correspondence between molecular sequences and computational paths. Thus the counting versions of NPpaths($s(n)$) and MOL-A($s(n)$) also coincide.

4.3. Oracles separating NPinit($s(n)$) from NPpaths($s(n)$)

We conjecture that NPinit($s(n)$) $\subset$ NPpaths($s(n)$). Let $f(n) = o(2^{2n}/n^k)$ for every k. To support our conjecture, we construct an oracle A such that NPpaths$(2^{n+1}-1)^A \not\subseteq$ NPinit$(f(n))^A$. For each set A, define

$$L^A = \{0^n : (\exists x, y)[|x| = |y| = n, x \in A, \text{ and } xy \in A]\}.$$

We construct A via an initial segment argument. We will ensure that A contains at most one string of length n for each n, so $L^A \in$ NPpaths$(2^{n+1}-1)$. Let $M_1, M_2, \ldots$ be an enumeration of NPinit($f(n)$) Turing machines such that M_e runs in time $n^e + e$.

Stage 0 Let $A = \emptyset$ and $n_0 = 0$.

Stage $e > 0$ Choose an odd number $n = n_e > 2n_{e-1}$ such that

$$f(n)(n^e + e)(n^e + e - 1) < 2^{2n}.$$

If there is some $B \subseteq \{0,1\}^{2n}$ such that $M_e^{A \cup B}(0^n)$ accepts then let $A = A \cup B$, defeating M_e, and go to stage $e+1$. Otherwise continue stage e. There must be a string x of length n such that $M_e(0^n)$ queries x on at most $f(n)(n^e + e)/2^n$ paths. Let $A = A \cup \{x\}$. If $M_e^A(0^n)$ accepts now, then we do nothing more, defeating M_e, and go to stage $e+1$. Otherwise continue stage e. The paths that query x can query at most $f(n)(n^e + e)(n^e + e - 1)/2^n < 2^n$ other strings. Let y be a string of length n such that no path queries both x and xy. Let $A = A \cup \{xy\}$, defeating M_e, and go to stage $e+1$.

Thus $L^A \in \mathrm{NPpaths}(2^{n+1} - 1)^A - \mathrm{NPinit}(f(n))^A$.

Define

$$\begin{aligned} L_{g,k}^A = \{0^n : (\exists x_1, \ldots, x_{n^k})[(|x_1| = \cdots = |x_{n^k}| = g(n)) \\ \text{and } (1 \le i \le n^k \Rightarrow x_1 \cdots x_i \in A)]\}. \end{aligned}$$

If $g(n) \neq O(\log n)$ then, by methods similar to those above, we can construct an oracle A such that

$$L_{g,k}^A \in \mathrm{NPpaths}(n^k 2^{g(n)})^A - \mathrm{NPinit}(2^{n^k})^A.$$

In particular, taking $g(n) = \log n(\log^* n - 1)$ and $k = 1$, we have an oracle A such that

$$\mathrm{NPpaths}(n^{\log^* n})^A \not\subseteq \mathrm{NPinit}(2^n)^A.$$

Chapter 5.

Molecular Approximation Algorithms for NP Optimization Problems

Molecular computers were introduced by Adleman [1, 36], but so far the field lacks a "killer application." It is well known that a DNA computer can solve SAT in linear time [36], but using an exponential number of DNA strands. It is essential to find applications of DNA computers that use subexponential volume.

In this paper we apply DNA computers to the approximate solution of NP-optimization problems. We show how DNA algorithms using subexponential volume and polynomial time can achieve much better accuracy than classical polynomial time approximation algorithms. We illustrate our method on Hochbaum and Maass's classical approximation algorithm [30] for the Covering problem. Our molecular algorithm runs in polynomial time and uses $O\left(\ell^2(\log \ell)n^2\binom{n(n-1)}{\ell^2/2}\right)$ volume to achieve error ratio $(1+\frac{1}{\ell})^2$. This algorithm does not use controversial operations like Intersection or even Amplify.

We also improve their algorithm in the classical computation model and implement it in a more complicated molecular computation model using the Amplify operation. Its volume is decreased to $O\left(c^{\ell^2/2}\ell^2(\log \ell)n^2\binom{n(n-1)}{\ell^2/2}\right)$, where $c = 0.823$. This is the first example we know in which the Amplify operation seems to enable DNA computers to solve a problem more efficiently.

Our results suggest a volume–accuracy tradeoff in DNA-based approximation algorithms for NP optimization problems. We think that approximation algorithms may be the "killer application" that will motivate the construction of DNA computers.

5.1. Hochbaum and Maass's Approximation Algorithm and Molecular Implementation

In this section we give a description of Hochbaum and Maass's approximation algorithm for the Opt-covering problem [30] and its implementation via a molecular computation.

Definition 26.

- Covering *problem:* Given a set P of points in the plane, a positive integer m, and a real number D, is it possible to cover all points of P with m disks of diameter D.
- Opt-covering *problem:* Given a set P of points on the plane and a real number D, find the minimum number of disks of diameter D that cover all points in P.
- An approximation A for Opt-covering has *error ratio* $1+\epsilon$ if
$$1 \leq \left|\frac{A(P,D)}{\mathrm{OPT}(P,D)}\right| \leq 1+\epsilon$$
for all P, where $\mathrm{OPT}(P,D)$ is the least number of disks covering all points in P.

The size of a problem instance is the number of points |P|, which we denote by n.

5.1.1. Hochbaum and Maass's Approximation Algorithm

Definition 27. Let H be an $aD \times bD$ rectangle with $b \leq a$, l be an integer and P be a set of points on plane.

- Partition H into $D \times bD$ strips: $I_1, I_2, \ldots, I_a$. For each $1 \leq i \leq \ell$, define
$$t_i = \begin{cases} 1 + \lfloor \frac{a-i}{\ell} \rfloor & \text{if } \lfloor \frac{a-i}{\ell} \rfloor = \frac{a-i}{\ell} \\ 2 + \lfloor \frac{a-i}{\ell} \rfloor & \text{otherwise} \end{cases}$$
$$H_{i,k} = \begin{cases} \bigcup_{j=1}^{i} \{I_j\} & \text{if } k = 1 \\ \bigcup_{j=(k-2)\ell+i+1}^{(k-1)\ell+i} \{I_j\}, & \text{if } 2 \leq k < t_i \\ \bigcup_{j=(t_i-2)\ell+i+1}^{a} \{I_j\} & \text{if } k = t_i \end{cases}$$

Each $H_{i,j}$ is called an *ℓ-strip* of H.

- For $i = 1, \ldots, \ell$, $\mathrm{P}_i(H) = \{H_{i,1}, \ldots, H_{i,t_i}\}$ is an *ℓ-partition* of H. Clearly each $\mathrm{P}_i(H)$ is a partition of H, induced by grouping ℓ small $D \times bD$-strips into a big $lD \times bD$ strip (boundary strips may be smaller).

- An *isolated* point is a point in P that has distance greater than D from any other point in P.

- $\mathrm{OPT}(H)$ is the least number of disks to cover all points of P in H.

- Fix Q as an optimal cover all points of P in H, $\mathrm{OPT}_i(H)$ is the number of disks in Q touching two adjacent l-strips in $\mathrm{P}_i(H)$.

Assume A is an approximation of Opt-covering with error ratio r_A. For each $\mathrm{P}_i(H)$, let

$$h_i = \sum_{E \in \mathrm{P}_i(H)} A(E) \leq r_A \cdot \sum \mathrm{OPT}(E) = r_A \cdot (\mathrm{OPT}(H) + \mathrm{OPT}_i(H)).$$

Since no disk in Q can contribute to both $\mathrm{OPT}_i(H)$ and $\mathrm{OPT}_j(H)$ with $i \neq j$,

$$\sum_{1 \leq i \leq l} \mathrm{OPT}_i(H) \leq \mathrm{OPT}(H).$$

$$\text{So, } h_1 + \cdots + h_l \leq r_A \cdot (l \cdot \mathrm{OPT}(H) + \sum_{i=1}^{l} \mathrm{OPT}_i(H)) \leq r_A \cdot (l+1) \cdot \mathrm{OPT}(H).$$

$$\text{Hence, } \min_{1 \leq i \leq l} h_i \leq r_A \cdot (1 + \frac{1}{l}) \cdot \mathrm{OPT}(H).$$

We have a new approximation for $\mathrm{OPT}(H)$, with error ratio $r_A \cdot (1 + \frac{1}{l})$, from an approximation to $\mathrm{OPT}(E)$ for $E \in \mathrm{P}_i(H)$ with error ratio r_A. Suppose all points of P are in area I, a $uD \times vD$ rectangle. The strategy is used in both a vertical and horizontal direction. I has partitions $\mathrm{P}_1(I), \ldots, \mathrm{P}_l(I)$ and each element $E \in \mathrm{P}_i(I)$ is a $l'D \times vD$ rectangle ($l' \leq l$). For each $E \in \mathrm{P}_i(I)$, E has partitions $\mathrm{P}_1(E), \ldots, \mathrm{P}_l(E)$. Each element $F \in \mathrm{P}_i(E)$ is a $l_1 D \times l_2 D$ rectangle ($l_1, l_2 \leq l$). Use a brute-force method to compute $\mathrm{OPT}(F)$ (each isolated point is covered with a disk and each of other disks has two points on its boundary). Since $(l/\sqrt{2})^2 = l^2/2$ disks can cover a whole $lD \times lD$ rectangle, the number of cases for the brute-force method is bounded by $\binom{n(n-1)}{l^2/2}$. The algorithm is formally described as follows; it has an error ratio of $(1+1/l)^2$.

Algorithm

Input P;

find a smallest $uD \times vD$ rectangle I to include all points in P;

for each $1 \le i \le l, E \in \mathrm{P}_i(I), 1 \le j \le l, F \in \mathrm{P}_j(E)$ and $F \cap P \ne \emptyset$ do

$a_F = \mathrm{OPT}(F)$ (via brute force method);

for each $1 \le i \le l, E \in \mathrm{P}_i(I)$, and $1 \le j \le l$ do

$b_{E,j} = \sum_{F \in \mathrm{P}_j(E) \text{ and } F \cap P \ne \emptyset} a_F$;

for each $1 \le i \le l, E \in \mathrm{P}_i(I)$ and $E \cap P \ne \emptyset$ do $b_E = \min_{1 \le j \le l} b_{E,j}$;

for each $1 \le i \le l$ do $c_{I,i} = \sum_{E \in \mathrm{P}_i(I) \text{ and } E \cap P \ne \emptyset} b_E$;

$c_I = \min_{1 \le i \le l} c_{I,i}$;

output(c_I);

End of Algorithm

5.1.2. Molecular Algorithm

Theorem 28. Covering $\in$ NPinit$\left(\binom{n(n-1)}{m}\right)$, *where* $n = |\mathrm{P}|$.

Proof: Let $\langle P, m\rangle$ be the input for Covering. Let M be the set of all isolated points in P. Let $P' = P - M$. For each isolated point, use one disk to cover it. Let $m' = m - |M|$ and $n' = |P'|$. In order to cover the points in P' we only consider to put each disk with two points at it boundary. Consider all of the $\binom{n'(n'-1)}{m'}$ arrangements, $\langle P, m\rangle \in$ Covering iff one of them can cover all points in P'.

It is easy to see there is a map g between the $\left\{\binom{n'(n'-1)}{m'}\right\}$ arrangements and $\left\{1, \ldots, \binom{n'(n'-1)}{m'}\right\}$ such that g is 1-1 and polynomial time computable. Nondeterministically choose an integer $i \le \binom{n(n-1)}{m}$, if $i > \binom{n'(n'-1)}{m'}$, then return *reject*; otherwise, return *accept* iff the arrangement $g(i)$ can cover all points in P'. ∎

Proposition 29.

i. Opt-covering *can be computed in polynomial time via* $|P|$ *parallel queries to* Covering *and for each query* $\langle P, m\rangle$, $m \leq n$.

ii. Opt-covering *can be computed in polynomial time via* $\log_2|P|$ *sequential queries to* Covering *and for each query* $\langle P, m\rangle$, $m \leq n$.

Proof: i. Opt-covering(P) is equal to the smallest $i \leq |P|$ such that $(P, i) \in$ Covering.

ii. Use binary search to find the smallest i above via query Covering.

∎

Theorem 30.

i. There is an approximation algorithm with error ratio $(1+\frac{1}{\ell})^2$ *for* Opt-covering *that runs in* $n^{O(1)}$ *time and makes* $\frac{\ell^2}{2}(\ell\cdot n)^2$ *parallel queries to* Covering *where for each query* $\langle M, m\rangle$, $|M| \leq n$ *and* $m \leq \frac{\ell^2}{2}$.

ii. There is an approximation algorithm with error ratio $(1+\frac{1}{\ell})^2$ *for* Opt-covering *that runs in* $n^{O(1)}$ *time and makes* $(\log\frac{\ell^2}{2})(\ell\cdot n)^2$ *adaptive queries to* Covering *where for each query* $\langle M, m\rangle$, $|M| \leq n$ *and* $m \leq \frac{\ell^2}{2}$

Proof: By Hochbaum and Maass's algorithm and Proposition 29. ∎

We are going to transform the approximation algorithm into a molecular algorithm.

Theorem 31. Covering $\in$ MOL($\binom{n(n-1)}{m}$).

Proof: Follows from Theorem 10 and Theorem 28. ∎

Definition 32. A MOL$*(s(n))$ algorithm is an algorithm with Init, Test, Separate, Append and Merge that runs polynomial steps and uses no more than $s(n)$ DNA strands.

Theorem 33. *For each* $\ell > 0$, *there is a* MOL$*(O(\ell^2(\log\ell)n^2\binom{n(n-1)}{\ell^2/2}))$ *algorithm of error ratio* $(1+\frac{1}{\ell})^2$ *for* Opt-covering.

Proof: By Theorem 31 and Theorem 30 and Hochbaum and Maass's algorithm.

∎

5.2. Improving Hochbaum and Maass's Algorithm

In order to decrease the volume needed for molecular computation, we improve Hochbaum and Maass's algorithm in the classical computation model. The most expensive part of their algorithm is at the brute-force method in finding the optimal solution for Opt-covering. Since Opt-covering can be computed via querying Covering, we focus on improving the algorithm for Covering.

Theorem 34. Covering $\in$ NPpath $\left(c^m \binom{n(n-1)}{m}\right)$ *, where* $c = 0.823$ *and* $n = |P|$ *for input* $\langle P, m \rangle$*.*

Proof: We use a new nondeterministic algorithm to determine Covering. A *position* is either an isolated point of P or else a pair of non isolated points of P with one more bit that determines the direction to put a disk with the two points at its boundary. Clearly the number of positions of P is bounded by $n(n-1)$. If P can be covered by the m disks, there is a position where a disk can cover at least $\frac{|P|}{m}$ points of P. The size of the problem can be quickly decreased after a disk is put on the position.

function Covering(P, m)
    unmark all of the positions;
    for $i = 1$ to $n(n-1)$ do
        if $P = \emptyset$ then return($accept$);
        if $m = 0$ then return($reject$);
        find an unmarked position p where a disk can cover at least $\frac{|P|}{m}$ points;
        mark the position p;
        nondeterministically choose $a_i \in \{0, 1\}$;
        if $a_i = 1$ then
            put a disk at position p;
            let M be the set of all points covered by the disk above;
            $P = P - M$;
            $m = m - 1$;

mark all positions that are adjacent to any point in M;

The algorithm is a nondeterministic. Every computation path corresponds to a string $a_1\cdots a_{n(n-1)}$ (append some 0s if it stops before $i = n(n-1)$) with $\sum_{i=1}^{n(n-1)} a_i \le m$. Let $\alpha = 0.198, \beta = 1.0 - \alpha$.

Lemma 35. *If $2m \le k$, there are at most $0.5c^m\binom{k}{m}$ strings in the set*

$$\{a_1\cdots a_k : (a_1 + \cdots + a_{k/2} \le \alpha m) \wedge (a_1 + \cdots + a_k \le m)\}.$$

Proof: The number of strings in

$$\{a_1\cdots a_k : (a_1 + \cdots + a_{k/2} \le \alpha m) \wedge (a_1 + \cdots + a_k \le m)\}$$

is at most

$$\begin{aligned} m\sum_{i=0}^{\alpha m}\binom{k/2}{i}\binom{k/2}{m-i} &\le m(\alpha m+1)\binom{k/2}{\alpha m}\binom{k/2}{\beta m} \\ &= \frac{1}{(2-o(1))^m(\alpha^\alpha\beta^\beta)^m}\binom{k}{m} \\ &\le 0.5c^m\binom{k}{m}. \end{aligned}$$

∎

Lemma 36. *There are at most $0.5c^m\binom{n(n-1)}{m}$ paths $a_1\cdots a_{n(n-1)}$ with the property $a_1 + \cdots + a_{n(n-1)/2} \ge \alpha m$.*

Proof: For every path $a_1\cdots a_{n(n-1)}$ with the property $a_1 + \cdots + a_{n(n-1)/2} \ge \alpha m$, it has used at least αm disks before $i = n(n-1)/2$ in the algorithm. After $i = n(n-1)/2$, there are at most

$$(1-\frac{1}{m})(1-\frac{1}{m-1})\cdots(1-\frac{1}{m-\alpha m+1})n = \beta n$$

points left in P and the number of unmarked positions is $\le (\beta n)^2$. So, the total number of positions chosen in the path is at most $\frac{n(n-1)}{2}+(\beta n)^2 \le (\frac{1}{2}+\beta^2+\frac{2}{n})n(n-1)$. The number of paths of this kind is at most

$$m\binom{(\frac{1}{2}+\beta^2+\frac{2}{n})n(n-1)}{m} \le m(\frac{1}{2}+\beta^2+\frac{2}{n})^m\binom{n(n-1)}{m} \le 0.5c^m\binom{n(n-1)}{m}.$$

∎

For $m > |P|$, it is trivial. We can assume $m \leq |P|$. Combining the two lemmas above we get the theorem.

∎

Definition 37. A MOL-A$*(s(n))$ algorithm is an algorithm with Separate, Merge, Append, Amplify, Init(1) and Test operations.

Theorem 38. Covering $\in$ MOL-A $\left(c^m \binom{n(n-1)}{m}\right)$, *where* $c = 0.823$.

Proof: By Theorem 25 and Theorem 34.

∎

Theorem 39. *For each* $\ell > 0$, *there is a* MOL-A$*$ $\left(O(\ell^2(\log \ell)n^2 c^{\ell^2/2}\binom{n(n-1)}{\ell^2/2}))\right)$ *algorithm of error ratio* $(1+\frac{1}{\ell})^2$ *for* Opt-covering, *where* $c = 0.823$.

Proof: By Theorem 38 and Theorem 30 and Hochbaum and Maass's algorithm.

∎

Chapter 6.

Counting and Random Computation with a Molecular Computer

In [46, 6], they designed DNA algorithm to solve a PSPACE-complete problem, but they used the operation that requires the strands of DNA anneal with their exact counterparts, in parallel for all different strands in the test tube. This is biologically impractical. Since there are many DNA sequences, it is very hard for one DNA sequence to meet another one that is the same as it.

Given a boolean formula f, #SAT(f) is the number of satisfiable assignments of f. #SAT plays a very important role in complexity theory. The polynomial time hierarchy can be reduced to #SAT via polynomial time Turing reductions [57]. We show that there is a molecular algorithm for #SAT and only use practical operations. It greatly improves the power of molecular computer with practical operations.

It is well known that a DNA computer can solve SAT in linear time [36], but using an exponential number of DNA strands. Although recent papers [3, 39, 9] solve NP problems using smaller exponential volume, we believe that it is essential to find applications of DNA computers that use subexponential volume. We apply a DNA computer with subexponential volume in random computation. It can improve the power of probabilistic computation.

6.1. Comparing the Number of DNA Sequences in Two Test Tubes

In this section we design a molecular algorithm to compare the number of DNA sequences in two test tubes. Before giving the molecular algorithm, let's get some intuition from the following simple algorithm.

6.1.1. Comparing the Number of Men and Women

There are two groups of people. One group consists a large number of men and the another group consists a large number of women. We need to determine which group has more people.

We merge the two groups. Let each man find a woman randomly as his wife. Separate all of those couples. We can determine if the number of men is greater than women from those persons left.

We use the same idea to compare the number of DNA sequences in two tubes T_1, T_2. Append a DNA sequence s to T_1 and the reverse of its complement $\overline{s}^{\wedge}$ to T_2. Merge the two tubes. Sequences of T_1 will bind to sequences of T_2 via s and $\overline{s}^{\wedge}$. This looks like a man finding a woman as his wife randomly. Separate all double sequences. We know which tube has more sequences from those sequences left.

Definition 40.

- For $D \in \{A, T, G, C\}$, D^ℓ is a DNA sequence of s D characters.
- $\updownarrow A^\ell$ represents a double DNA sequence combined by A^ℓ and T^ℓ.
- $\begin{bmatrix} A \\ T \end{bmatrix}^\ell$ is the set of all sequences $A^{\ell_1}(\updownarrow A^{\ell_2})T^{\ell_3}$ such that $s_2 > 0$ and $\ell_1 + \ell_2 = \ell_2 + \ell_3 = \ell$.
- For $y = y_1 \cdots y_m$, $y^\wedge$ is the reverse of string y: $y_m \cdots y_1$.
- Let T be a test tube of DNA sequences. $|T|$ is the number of sequences in T. $l(T)$ is the largest length of DNA sequences in T.
- For a set $S \subseteq \Sigma^*$ and $x \in \Sigma^*$, $S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$

- For a DNA sequence $s \in \{A, T, G, C\}$, $\overline{s}$ is the complement of s. For example, if $s = AGTA$, then $\overline{s} = TCAT$.

6.1.2. A Simple Algorithm for Comparison

Hypothesis 1: There is a constant w_0 such that for any two DNA sequences xs and $y\overline{s}^\wedge$ with $|s| = w_0$, they can anneal to form $x(\updownarrow s)y^\wedge$ when they meet.

The following operation as well as the algorithm is based on this assumption.

Anneal(T_1, T_2, T_3)

Before the operation:

$T_1 = \{x_1 s, \ldots, x_{m_1} s\}$;

$T_2 = \{y_1 \overline{s}^\wedge, \ldots, y_{m_2} \overline{s}^\wedge\}$;

$T_3 = \emptyset$;

$|s| = 30$.

After the operation:

$$
\begin{aligned}
T_1 &= T_2 = \emptyset; \\
m_1 < m_2 \implies & T_3 = \{x_1(\updownarrow s)y_{i_1}^\wedge, \ldots, x_{m_1}(\updownarrow s)y_{i_{m_1}}^\wedge, y_{j_1}s, \ldots y_{j_{m_2-m_1}}s\}, \\
& \text{where } \{i_1, \ldots, i_{m_1}\} \cup \{j_1, \ldots, j_{m_2-m_1}\} = \{1, \ldots, m_2\}. \\
m_1 = m_2 \implies & T_3 = \{x_1(\updownarrow s)y_{i_1}^\wedge, \ldots, x_{m_1}(\updownarrow s)y_{i_{m_1}}^\wedge\}, \\
& \text{where } i_1, \ldots, i_{m_1} \text{ is a permutation of } 1, \ldots, m_2. \\
m_1 > m_2 \implies & T_3 = \{x_{i_1}(\updownarrow s)y_1^\wedge, \ldots, x_{i_{m_2}}(\updownarrow s)y_{i_{m_2}}^\wedge, x_{j_1}s, \ldots x_{j_{m_1-m_2}}s\}, \\
& \text{where } \{i_1, \ldots, i_{m_2}\} \cup \{j_1, \ldots, j_{m_1-m_2}\} = \{1, \ldots, m_1\}.
\end{aligned}
$$

We believe the Anneal is a practical operation. Assume $|T_1| < |T_2|$. For a sequence in T_1, it is easy to find another sequence in T_2 to anneal after T_1, T_2 are merged. Suppose only 20 percent of the sequences in T_1 can find sequences to anneal with. We separate all the annealed sequences from the test tube. The sequences left will continue to anneal. Repeat the procedure. Eventually all sequences of T_1 will combine with sequences in T_2. The number of steps is only $O(\log |T_1|)$ if every time we have 20 percent of the sequences annealed.

function Compare(T_1, T_2)

find two $s_1 \neq s_2$ DNA sequences;

Append(T_1, s_1);

Append(T_2, s_2);

find a DNA sequence $s \in \{A, T, G, C\}^{w_0}$;

Append(T_1, s);

Append($T_2, \overline{s}^{\wedge}$);

Anneal(T_1, T_2, T_0);

Separate all double sequence in T;

If there a single sequence of T_1 left in T then return $|T_1| > |T_2|$

else if there a single sequence of T_2 left in T then return $|T_1| < |T_2|$

else return $|T_1| = |T_2|$;

The two DNA sequences s_1, s_2 are used to distinguish the sequences in T_1 with those in T_2. s_1 and s_2 should be chosen so that for any DNA sequence d in T_1 or T_2, neither s_1 nor s_2 has long common subsequence with d or $\overline{d}$. Since d usually encodes of $0, 1$ sequences, it is not hard to find s_1, s_2 (we can let each $d \in \{A, T\}^*$ and let $s_1, s_2 \in \{G, C\}^*$). There may exist several ways to separate the partial double sequences and check if T has any sequence from T_1 left. The following is one of the them.

(1) Sep(T_0, s_1, T_3, T_4);

(2) Sep(T_3, s_2, T_5, T_6);

(3) if Not Test(T_6) then $|T_1| > |T_2|$

(4) else

(5) if Not Test(T_4) then $|T_1| < |T_2|$

(6) else $|T_1| = |T_2|$;

After (2), T_5 contains all partial double sequences $(d_1(\updownarrow s)d_2)$ that are the combination of a sequence from T_1 and a sequence from T_2, T_4 contains all sequences in T_2 that cannot combine with any sequence in T_1, and T_6 contains all sequences in T_1 that cannot combine with any sequence in T_2.

Theorem 41. *Under the assumption of Hypothesis 1, there is an { Append, Separate,* *Anneal, Test } molecular algorithm to compare the number of DNA sequences in two test tubes.*

6.1.3. Revised Algorithm for Comparison

In order to make our operation more practical, we let our anneal operation base on the following weaker assumption. We use c to represent a constant as well as the constant function $c(x) = c$..

Hypothesis 2: There is an easily computable function $m(n_1, n_2)$ from $\mathsf{N} \times \mathsf{N}$ to N such that for two DNA sequences xA^ℓ and yT^ℓ, if $s = m(|x|, |y|)$, they will anneal to form $xSy^\wedge$ for some $S \in \begin{bmatrix} A \\ T \end{bmatrix}^\ell$ when they meet.

Anneal$_c(T_1, T_2, T_3)$

Before the operation:

$T_1 = \{x_1A^\ell, \ldots, x_{m_1}A^\ell\}$;

$T_2 = \{y_1T^\ell, \ldots, y_{m_2}T^\ell\}$;

$T_3 = \emptyset$.

$|x_1| = |x_2| = \cdots = |x_{m_1}|$ and $|y_1| = |y_2| = \cdots = |y_{m_2}|$ and $s = m(|x_1|, |y_1|)$.

After the operation:

$$T_1 = T_2 = \emptyset;$$

$$\begin{aligned} m_1 + c < m_2 \implies & T_3 = \{x_1S_1y_{i_1}^\wedge, \ldots, x_{m_1}S_{m_1}y_{i_{m_1}}^\wedge, y_{j_1}T^\ell, \ldots, y_{j_{m_2-m_1}}T^\ell\}, \\ & \text{where } \{i_1, \ldots, i_{m_1}\} \cup \{j_1, \ldots, j_{m_2-m_1}\} = \{1, \ldots, m_2\} \\ & \text{and } S_i \in \begin{bmatrix} A \\ T \end{bmatrix}^\ell \text{ for } i = 1, \ldots, m_1. \end{aligned}$$

$$\begin{aligned} m_1 > m_2 + c \implies & T_3 = \{x_{i_1}S_1y_1^\wedge, \ldots, x_{i_{m_2}}S_{m_2}y_{i_{m_2}}^\wedge, x_{j_1}A^\ell, \ldots, x_{j_{m_1-m_2}}A^\ell\}, \\ & \text{where } \{i_1, \ldots, i_{m_2}\} \cup \{j_1, \ldots, j_{m_1-m_2}\} = \{1, \ldots, m_1\} \\ & \text{and } S_i \in \begin{bmatrix} A \\ T \end{bmatrix}^\ell \text{ for } i = 1, \ldots, m_2. \end{aligned}$$

$$|m_1 - m_2| \le c \implies \text{No result .}$$

Lemma 42. *There is a $O(l(T))$ time { Separate, Merge, Test } molecular algorithm such that for any test tube T of DNA sequences that only contains DNA sequences encoding sequences in Σ^*, it outputs $l(T)$ and a series of test tubes $T_1, \ldots, T_k$ such that each T_j contains all sequences of T in Σ^j.*

Proof:

function $l(T)$

$i = 1$;

while (Not Test(T)) do

Separate($T, 0, i, T', T''$);

Separate($T'', 1, i, T''', T_i$);

Merge(T', T''', T);

$i = i + 1$;

output($30(i-1), T_1, \ldots, T_i$);

It is easy to see that T_i contains all sequences of in T of Σ^i. It also outputs the largest length $30(i-1)$ of sequences in T since we use 30 A, T, G, C characters to encode one $0, 1$ bit. ∎

Lemma 43. *Assume T only contains DNA sequences encoding Σ^*. There is a $O(l(T))$ time {Append, Separate, Merge} molecular algorithm such that for any test tube of DNA sequences, it outputs another test tube T' of DNA sequences with the following properties: 1) T' has the same number of DNA sequences as T; and 2) all sequences in T' have the same length.*

Proof: Use the Separate operation to partition the DNA sequences into different tubes so that all sequences in the same tube are of the same length. Append some sequences at the end of sequences of small length. Merging all tubes we can get a tube that has the same number of sequence as before and all sequences have the same length. The algorithm is as follows: From the proof of Lemma 42, we can get $T_1, \ldots, T_k$ such that T_i contains all sequences of T in Σ^i. Let T' be empty.

$k = \frac{l(T)}{30}$;

for $i = 1$ to $k - 1$ do

for $j = i$ to $k - i$ do Append$(T_i, 0)$;

for $i = 1$ to k do Merge(T', T_i, T'); ▮

Lemma 44. *There is a* $O(m(l(T_1), l(T_2)))$ { *Append, Anneal$_c$, Separate, Test* } *molecular algorithm such that for two test tubes* T_1 *and* T_2 *with* $||T_1| - |T_2|| > c$, *it correctly outputs* $|T_1| > |T_2|$ *or* $|T_1| < |T_2|$.

Proof: We just set $|s|$ to be $m(l(T_1), l(T_2))$ and replace Anneal by Anneal* in Compare. ▮

In the proof of Lemma 44, we could also consider using Separation by length to separate all of the partial double sequences. This may make it easier to implement.

Theorem 45 follows from Lemma 42, 43, 44.

Theorem 45. *There is a* {*Append, Merge, Separate, Anneal$_c$, Test* } *molecular algorithm Compare$_c$ such that given two test tubes of DNA sequences encoding sequences in* Σ^*, *it correctly outputs* $|T_1| > |T_2| + c$ *or* $|T_1| + c < |T_2|$ *in* $O(\max\{l(T_1), l(T_2)\}, m(l(T_1), l(T_2))$ *time.*

at T_1 and T_2 respectively,

6.2. Counting

In this section we design a molecular algorithm to compute PP. The idea is to generate all satisfiable assignments and all unsatisfiable assignments in the two test tubes. It can be done by Init operation and Lemma 9. Using the techniques in the last section, we know which tube contains more sequences.

Definition 46.

- $L \in$ PPinit$[s(n)]$ if there is an NPinit$[s(n)]$ Turing machine such that, for all x, $x \in L$ iff the number of accepting paths is greater than the number of rejecting paths.

- $L \in$ PPinit$'[d(n), s(n)]$ if there is an NPinit$[s(n)]$ Turing machine such that, for all x,

- $x \in L \Rightarrow$ the number of accepting paths is greater than the number of rejecting paths plus $d(n)$ and
- $x \notin L \Rightarrow$ the number of accepting paths is less than the number of rejecting paths minus $d(n)$.

- $L \in$ PPpath$[s(n)]$ if there is an NPpath[s(n)] Turing machine such that, for all x, $x \in L$ iff the number of accepting paths is greater than the number of rejecting paths.

- $L \in$ PPpath$'[d(n), s(n)]$ if there is an NPpath[s(n)] Turing machine such that, for all x,
  - $x \in L \Rightarrow$ the number of accepting paths is greater than the number of rejecting paths plus $d(n)$ and
  - $x \notin L \Rightarrow$ the number of accepting paths is less than the number of rejecting paths minus $d(n)$.

- A PP-MOL$[s(n)]$ computation starts with a Init$(T_0, s(n))$ operation. It permits Append, Separate and Merge in the middle of the computation. The final operation is always Compare(T', T''). The time is bounded by a polynomial in the length of the input. It accepts if the output of Compare is $|T'| > |T''|$.

- $L \in$ PP-MOL$[s(n)]$ if the there is a polynomial time uniform family of PP-MOL$[s(n)]$ computations that accept L.

- A PP-MOL$'[c, s(n)]$ computation starts with a Init$(T_0, s(n))$ operation. It permits Append, Separate and Merge in the middle of the computation. The final operation is always Compare$_c(T', T'')$. The time is bounded by a polynomial in the length of the input. It accepts if the output of Compare$_c$ is $|T'| > |T''| + c$.

- $L \in$ PP-MOL$'[c, s(n)]$ if the there is a polynomial time uniform family of PP-MOL$'[c, s(n)]$ computations that accept L.

- A PP-MOL-A$[s(n)]$ computation starts with a tube T_0 of one empty DNA sequence. It permits Append, Separate, Merge and Amplify operations in the middle of the computation. Final operation is Compare(T', T''). The total number of DNA sequences used is no greater than $s(n)$. The time is bounded by a polynomial of the length of input. It accepts if the output of Compare is $|T'| > |T''|$.

- $L \in$ PP-MOL-A$[s(n)]$ if the there is a polynomial time uniform family of PP-MOL-A$[s(n)]$ computations that accept L.
- A PP-MOL-A$'[c, s(n)]$ computation starts with a tube T_0 of one empty DNA sequence. It permits Append, Separate, Merge and Amplify operations in the middle of the computation. The final operation is always Compare$_c(T', T'')$. The total number of DNA sequences used is no greater than $s(n)$. The time is bounded by a polynomial of the length of input. It accepts if the output of Compare$_c$ is $|T'| > |T''| + c$.
- $L \in$ PP-MOL-A$'[c, s(n)]$ if the there is a polynomial time uniform family of PP-MOL-A$'[c, s(n)]$ computations that accept L.

Lemma 47.

i. PPinit$[s(n)] \subseteq$ PP-MOL$[s(n)]$.

ii. PPinit$'[c, s(n)] \subseteq$ PP-MOL$'[c, s(n)]$.

Proof: (i). Let $L \in$ PPinit$[s(n)]$. There is a polynomial time uniform family of polynomial size circuits $\{\pi_n(x, y)\}_{i=1}^{\infty}$ such that for input of length n, $x \in L$ iff

$$|\{y : 1 \leq y \leq s(|x|) \text{ and } \pi_n(x, y) = 1\}| > |\{y : 1 \leq y \leq s(|x|) \text{ and } \pi_n(x, y) = 0\}|.$$

Let x be an input of length n. By Lemma 9, we can get two tubes T_1 and T_2 such that $T_1 = \{y : 1 \leq y \leq s(|x|) \text{ and } \pi_n(x, y) = 1\}$ and $T_2 = \{y : 1 \leq y \leq s(|x|) \text{ and } \pi_n(x, y) = 0\}$. By Lemma 45, we can determine if $x \in L$ via Compare(T_1, T_2).

(ii). It is very similar to the proof in (i). ∎

Lemma 48.

i. PP-MOL$[s(n)] \subseteq$ PPinit$[s(n)]$.

ii. PP-MOL$'[c, s(n)] \subseteq$ PPinit$'[c, s(n)]$.

Proof: (i). It is similar to the proof of Theorem 21. The first operation of a PP-MOL$[s(n)]$ computation is Init$(T_0, s(n))$. It produces $s(n)$ DNA sequences encoding $1, 2, \ldots, s(n)$. Construct an NPinit$(s(n))$ Turing machine such that path i simulates the DNA sequence that encodes i. If the final operation of the PP-MOL$[s(n)]$ computation is Compare(T', T''), all paths that simulate the DNA sequences in T' accept and all paths that simulate the DNA sequences in T'' reject.

(ii). It is very similar to the proof in (i). ∎

Theorem 49.

i. PPinit$[s(n)] =$ PP-MOL$[s(n)]$.

ii. PPinit$'[c, s(n)] =$ PP-MOL$'[c, s(n)]$

Proof: Follows from Lemma 47 and Lemma 48. ∎

Lemma 50.

i. PPpath$[s(n)] \subseteq$ PP-MOL-A$[s(n)]$.

ii. PPpath$'[c, s(n)] \subseteq$ PP-MOL-A$'[c, s(n)]$

Proof: It is similar to the proof of Lemma 24. Use one DNA sequence to simulate a partial path. When a partial path splits (nondeterministically) into several paths, use Amplify to generate the same number of DNA sequences. ∎

Lemma 51.

i. PP-MOL-A$[s(n)] \subseteq$ PPpath$[s(n)]$.

ii. PP-MOL-A$'[c, s(n)] \subseteq$ PPpath$'[c, s(n)]$.

Proof: It is similar to the proof of Lemma 23. Use one path to simulate a DNA sequence. When an Amplify operation is executed, use nondeterminism to deal with it. ▮

Theorem 52.

i. PPpath$[s(n)] =$ PP-MOL-A$[s(n)]$.

ii. PPpath$'[c, s(n)] =$ PP-MOL-A$'[c, s(n)]$

Proof: Follows from Lemma 50 and Lemma 51. ▮

Lemma 53.

i. PPinit$[s(n)] \subseteq$ PPinit$[d(n), d(n)(2s(n) + 1)]$.

ii. PPpath$[s(n)] \subseteq$ PPpath$'[d(n), d(n)(2s(n) + 1)]$.

Proof: Let NP TM M accept L. Let M' be a TM derived from M by splitting each accepting path into two accepting paths and each rejecting paths into two rejecting paths; and add one more rejecting path. We have $x \in L \Rightarrow$ the number of accepting paths of M' is greater than the number of rejecting path, and $x \notin L \Rightarrow$ the number of accepting paths of M' is less than the number of rejecting path. Let M'' be a TM derived from M' such that each accepting path is splitted into $d(n)$ accepting paths and each rejecting path is splitted into $d(n)$ rejecting paths. Clearly, if M is an NPinit TM, then so is M'', and if M' is an NPpath TM, then so is M''. ▮

Theorem 54.

i. PPinit$[s(n)] \subseteq$ PP-MOL$'[c, c(2s(n) + 1)]$.

ii. PPpath$[s(n)] \subseteq$ PP-MOL-A$'[c, c(2s(n) + 1)]$.

Proof: It follows from Lemma 53 and Theorem 52, and 49. ▮

Corollary 55.

i. There is an { Init, Append, Merge, Separate, Anneal, Test } molecular algorithm for #SAT *that runs in $n^{O(1)}$ time and 2^n volume.*

ii. There is an { Init, Append, Merge, Separate, Anneal$_c$, Test } molecular algorithm for #SAT *that runs in $n^{O(1)}$ time and $O(2^n)$ volume.*

Proof: (i). Let $L = \{\langle f, k\rangle : \#\mathrm{SAT}(f) \geq k\}$. It is easy to see that L is in PPinit$[2^{n+1}]$. Thus #SAT(f) can be determined via querying L and each query is of the form $\langle f, k\rangle$ for some $0 \leq k \leq 2^n$. By (i) of Theorem 49, we have the corollary.

(ii). It is very similar to (i). We use Theorem 54 instead of Theorem 49. ∎

Corollary 56. *For every language L in the polynomial time hierarchy, there is an { Init, Append, Merge, Separate, Anneal, Test } molecular algorithm to accept L which runs in $n^{O(1)}$ time and $2^{n^{O(1)}}$ volume.*

Proof: It follows from Toda's Theorem PH $\in \mathrm{P}_T^{\#\mathrm{SAT}}$ and Corollary 55 ∎

6.3. RP and RP-MOL

In this section we study the power of random computation via molecular computer with one-sided error. We design an algorithm to produce random DNA sequences that will be used in the random test. Since a test tube can contain a lot of DNA sequences, the power of the random computation can be improved.

We believe the following Rand-partition and Init-E are practical operations. We can imagine many of physical methods to partition a solution randomly.

Rand-partition(T, T_1, T_2)

Every sequence in T has probability $1/2$ to go to T_1 and T_2.

$T = \emptyset$;

Init-E(T, m)

$T = \{m$ copies of empty DNA sequences$\}$.

Lemma 57. *There is a $O(m)$ time { Rand-partition, Append, Merge } molecular algorithm to produce a random multi-subset of size s in Σ^m.*

Proof:

function Rand-init(T, s, m)

Init-E(T, s);

for $i = 1$ to s do

  Rand-partition(T, T_0, T_1);

  Append$(T_0, 0)$;

  Append$(T_1, 1)$;

  Merge(T_0, T_1, T);

return(T);

At each time in the loop, each sequence has 0.5 probability of having 0 or 1 appended to it. Each empty sequence in the initial T has $1/2^m$ chance to become any sequence of Σ^m. So, Rand-init(T, s, m) produces s random samples of Σ^m in test tube T. ∎

Definition 58.

- A RP-MOL$[s(n)]$ computation starts with a Rand-init(T, s, m) operation. It permits Append, Separate, and Merge in the middle of the computation. It accepts if T_0 is not empty at the end of the computation. The number of steps is bounded by a polynomial.

- $L \in$ RP-MOL$[s(n)]$ if L is accepted by polynomial time uniform family of RP-MOL$[s(n)]$ computations $\{M_x\}_{x \in \Sigma^*}$ such that for any $k > 0$,

$$\begin{aligned} x \in L &\implies M_x \text{ accepts } x \text{ with probability at least } 1 - 1/n^k \text{ for all large } n \\ x \notin L &\implies M_x \text{ accepts } x \text{ with probability } 0. \end{aligned}$$

The classical definition about RP is as follows.

Definition 59. $L \in$ RP if there is a set $A \in$ P and a polynomial $p(n)$ such that

$$\begin{aligned} x \in L &\implies \text{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \geq \frac{3}{4} \\ x \notin L &\implies \text{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] = 0. \end{aligned}$$

Proposition 60. *The following items are all equivalent.*

i. $L \in \mathrm{RP}$

ii. there are polynomials $p(n), q(n)$ *and a set* $A \in \mathrm{P}$ *such that*

$$\begin{aligned} x \in L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \geq \frac{1}{q(n)} \\ x \notin L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] = 0. \end{aligned}$$

iii. For any polynomial $q(n)$*, there is a polynomial* $p(n)$ *and a set* $A \in \mathrm{P}$ *such that*

$$\begin{aligned} x \in L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \geq 1 - \frac{1}{2^{q(n)}} \\ x \notin L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] = 0. \end{aligned}$$

Proof: We just prove the (ii) implies (iii). For input x of length n, let $k = p(n)q(n)$. Let $L_1 = \{\langle x, y_1 \cdots y_k\rangle : \langle x, y_i\rangle \in L \text{ for some } 1 \leq i \leq k\}$. Assume $x \in A$. For randomly chosen $y_1, \ldots, y_k$ from $\Sigma^{p(n)}$, the probability that $\langle x, y_1 \cdots y_k\rangle \in L_1$ is $\geq 1 - (1 - \frac{1}{q(n)})^k$. If $x \notin A$, we have the probability that $\langle x, y_1 \cdots y_k\rangle \in L_1$ is 0. ∎

Let us generalize the definition of RP as follows.

Definition 61. $L \in \mathrm{RP}[\frac{1}{g(n)}]$ if there is a polynomial $p(x)$ and a set $A \in \mathrm{P}$ such that

$$\begin{aligned} x \in L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \geq \frac{1}{g(n)} \\ x \notin L &\implies \mathrm{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] = 0. \end{aligned}$$

Theorem 62. *Let* $f(n)$ *be a nondecreasing function from* N *to* N*. If* $f(n) \neq O(\log n)$*, then* $\mathrm{RP}[\frac{1}{g(n)}] \subseteq \mathrm{RP\text{-}MOL}[c, g(n) \cdot f(n)]$.

Proof: Let $L \in \mathrm{RP}[\frac{1}{g(n)}]$ via $A \in \mathrm{P}$ and polynomial $p(n)$. Let $k = g(n) \cdot f(n)$. After executing Rand-init$(T, k, p(n))$. We have $T = \{y_1, \ldots, y_k\}$ that consists k random samples from $\Sigma^{p(n)}$. By Lemma 9, we can get a test tube T_0 that contains all of the y_i in T with $\langle x, y_i\rangle \in A$.

Case 1. $x \in L$.

$$\mathrm{Prob}[\{y_1, \ldots, y_k\} \cap \{y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}\} = \emptyset] \leq (1 - \frac{1}{g(n)})^k \approx \frac{1}{e^{k/g(n)}}.$$

So, $\text{Prob}[\{y_1, \ldots, y_k\} \cap \{y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}\} \neq \emptyset] \geq 1 - \frac{1}{e^{k/g(n)}}$.
With probability $\geq 1 - \frac{1}{e^{k/g(n)}}$, $T_0 \neq \emptyset$.
Case 2. $x \notin L$.

$$\text{Prob}[\{y_1, \ldots, y_k\} \cap \{y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}\} = 0.$$

So, the probability for $T_0 \neq \emptyset$ is 0.

∎

6.4. BPP and BP-MOL

In this section we study the power of random computation via a molecular computer with two-sided error. We produce random DNA sequences that will be used in the random test. We get two test tubes with positive result and negative result respectively. Comparing the number of DNA sequences in the two tubes, we can have a high probability of knowing the answer. Since a test tube can contain many DNA sequences, the power of the random computation can be improved.

Definition 63.

- A BP-MOL$[c, s(n)]$ computation starts with a Rand-init$(T, s(n), m)$ operation. It permits Append, Separate, and Merge in the middle of the computation. The final operation is always Compare(T_1, T_2). It accepts if the output of Compare$_c(T_1, T_2)$ is $|T_1| > |T_2| + c$. The number of steps is bounded by a polynomial.

- $L \in$ BP-MOL$[c, s(n)]$ if L is accepted by polynomial time uniform family BP-MOL$[c, s(n)]$ $\{M_x\}_{x\in\Sigma^*}$ computations such that for any $k > 0$,

$$x \in L \implies M_x \text{ accepts } x \text{ with probability at least } 1 - 1/n^k \text{ for all large } n.$$
$$x \notin L \implies M_x \text{ accepts } x \text{ with probability at most } 1/n^k \text{ for all large } n.$$

Definition 64. $L \in$ BPP if there is a set $A \in$ P and a polynomial $p(n)$ such that

$$x \in L \implies \text{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \geq \frac{3}{4}$$
$$x \notin L \implies \text{Prob}[y : \langle x, y\rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \leq \frac{1}{4}.$$

Theorem 65 (Hoeffding). *Let $Y_1, Y_2, \ldots, Y_n$ be independent random variables with zero means and ranges $a_i \leq Y_i \leq b_i$. For each $\eta > 0$*

$$\text{Prob}\{|Y_1 + \cdots + Y_m| > \eta| \leq 2e^{-\frac{2\eta^2}{\sum_{i=1}^{n}(b_i - a_i)^2}}$$

The proof of Theorem 65 can be found in Appendix B of [43].

Lemma 66. *Let $y_1, \ldots, y_k$ be independent random elements in set S. Let S_0, S_1 be a partition of S. For some $i \in \{0, 1\}$, $\text{Prob}[y_j \in S_i] \geq \frac{1}{2} + \epsilon$ for $1 \leq j \leq k$. We have $\text{Prob}[S_i(y_1) + \cdots + S_i(y_k) > k/2 + k\epsilon/2] > 1 - 2e^{-k\cdot\epsilon^2}$, where $S_i(y) = \begin{cases} 1 & \text{if } y \in S_i \\ 0 & \text{otherwise .} \end{cases}$.*

Proof:

$$D(y) = \begin{cases} 1 & \text{if } y \in S_i \\ -1 & \text{if } y \in S_{1-i} \end{cases}$$

The expectation of $D(y)$

$$\begin{aligned} \text{E}[D(y)] &\geq (\frac{1}{2} + \epsilon) \cdot 1 + (\frac{1}{2} - \epsilon) \cdot (-1) \\ &= 2\epsilon. \end{aligned}$$

Let $Y_i = D(y_i) - \text{E}[D(y)]$ and $\delta = k \cdot \epsilon$. By Theorem 65, we have

$$\text{Prob}[|Y_1 + \cdots + Y_k| > \delta] \leq 2e^{-2\cdot\delta^2/k} = 2e^{-k\epsilon^2}.$$

Therefore,

$$\text{Prob}[|Y_1 + \cdots + Y_k| \leq \delta] \geq 1 - 2e^{-k\epsilon^2}.$$

$|Y_1 + \cdots + Y_k| \leq \delta$ implies $k \cdot \text{E}[D(y)] - \delta \leq \sum_{i=1}^{k} D(y_i)$. Therefore, $k\epsilon < \sum_{i=1}^{k} D(y_i)$. Clearly,

$$\begin{aligned} \sum_{i=1}^{k} D(y_i) &= \sum_{i=1}^{k} S_i(y_i) - \sum_{i=1}^{k} S_{1-i}(y_i), \text{ and} \\ k &= \sum_{i=1}^{k} S_i(y_i) + \sum_{i=1}^{k} S_{1-i}(y_i) \end{aligned}$$

Therefore,

$$\text{Prob}[S_i(y_1) + \cdots + S_i(y_k) > S_{1-i}(y_1) + \cdots + S_{1-i}(y_k) + k\epsilon] > 1 - 2e^{-k\cdot\epsilon^2}.$$

So,

$$\text{Prob}[S_i(y_1) + \cdots + S_i(y_k) > k/2 + k\epsilon/2] > 1 - 2e^{-k\cdot\epsilon^2}.$$

∎

Proposition 67. *The following items are all equivalent.*

i. $L \in \text{BPP}$

ii. there are polynomials $p(n), q(n)$ and a set $A \in \text{P}$ such that

$$\begin{aligned} x \in L &\implies \text{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \geq \frac{1}{2} + \frac{1}{q(n)} \\ x \notin L &\implies \text{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \leq \frac{1}{2} - \frac{1}{q(n)}. \end{aligned}$$

iii. For any polynomial $q(n)$, there is a polynomial $p(n)$ and a set $A \in \text{P}$ such that

$$\begin{aligned} x \in L &\implies \text{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \geq 1 - \frac{1}{2^{q(n)}} \\ x \notin L &\implies \text{Prob}[y : \langle x, y\rangle \in A \textit{ and } y \in \Sigma^{p(|x|)}] \leq \frac{1}{2^{q(n)}}. \end{aligned}$$

Proof: We only prove that $(ii) \Rightarrow (iii)$. Let $y_1 \cdots y_k$ be k random strings in $\Sigma^{p(|x|)}$. Let

$$\begin{aligned} A_1 = \{\langle x, y_1 \cdots y_k\rangle : \quad & k = q(n)g(n)^2 \bigwedge \\ & y_i \in \Sigma^{p(n)} \text{ for } i = 1, \ldots, k \bigwedge \\ & \sum_{i=1}^{k} A(\langle x, y_i\rangle) > k/2\}. \end{aligned}$$

Fix an input x. Let

$$\begin{aligned} S &= \Sigma^{p(|x|)} \\ S_0 &= \{y : \langle x, y\rangle \notin A \wedge x \in S\} \\ S_1 &= \{y : \langle x, y\rangle \in A \wedge x \in S\} \\ \epsilon &= \frac{1}{g(n)} \end{aligned}$$

By the definition of BPP$[\frac{1}{g(n)}]$, we have that

$$\begin{aligned} x &\notin L \Rightarrow \text{Prob}[y_i \in S_0] \geq \frac{1}{2} + \epsilon, \text{ and} \\ x &\in L \Rightarrow \text{Prob}[y_i \in S_1] \geq \frac{1}{2} + \epsilon \end{aligned}$$

If $x \in L$, by Lemma 66, we have

$$\text{Prob}[\sum_{i=1}^{k} S_1(y_i) > k/2] \geq 1 - 2e^{-k\cdot\epsilon^2}.$$

Therefore,

$$\text{Prob}[\sum_{i=1}^{k} A(\langle x, y_i \rangle) > k/2] \geq 1 - 2e^{-k\cdot\epsilon^2}.$$

Hence,

$$\text{Prob}[\langle x, y_1 \cdots y_k \rangle \in A_1] \geq 1 - 2e^{-k\cdot\epsilon^2}.$$

If $x \notin L$, by Lemma 66, we have

$$\text{Prob}[\sum_{i=1}^{k} S_0(y_i) > k/2] \geq 1 - 2e^{-k\cdot\epsilon^2}.$$

Therefore,

$$\text{Prob}[\sum_{i=1}^{k} A(\langle x, y_i \rangle) < k/2] < 2e^{-k\cdot\epsilon^2}.$$

Hence,

$$\text{Prob}[\langle x, y_1 \cdots y_k \rangle \in A_1] < 2e^{-k\cdot\epsilon^2}.$$

∎

Let us generalize the definition of BPP.

Definition 68. $L \in$ BPP$[\frac{1}{g(n)}]$ if there is a polynomial $p(x)$ and a set $A \in$ P such that

$$\begin{aligned} x \in L &\Longrightarrow \text{Prob}[y : \langle x, y \rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \geq \frac{1}{2} + \frac{1}{g(n)} \\ x \notin L &\Longrightarrow \text{Prob}[y : \langle x, y \rangle \in A \text{ and } y \in \Sigma^{p(|x|)}] \leq \frac{1}{2} - \frac{1}{g(n)}. \end{aligned}$$

Theorem 69. *Let $f(n)$ be a nondecreasing function from N to N. If $f(n) \neq O(\log n)$, then* BPP$[\frac{1}{g(n)}] \subseteq$ BP-MOL$[g(n)^2 \cdot f(n)]$.

Proof: The proof is similar to that of $(ii) \Rightarrow (iii)$ in Proposition 67. Let $L \in \mathrm{BPP}[\frac{1}{g(n)}]$ via $A \in \mathrm{P}$ and polynomial $p(n)$. Let $k = g(n)^2 \cdot f(n)$. After executing Rand-init$(T, k, p(n))$. We have $T = \{y_1, \ldots, y_k\}$ that consists of k random samples from $\Sigma^{p(n)}$. By Lemma 9, we can get two test tubes $T_0 = \{y : \langle x, y\rangle \in A \wedge y \in T\}$ and $T_1 = \{y : \langle x, y\rangle \notin A \wedge y \in T\}$. Let

$$\begin{aligned} S &= \Sigma^{p(n)} \\ S_0 &= \{y : \langle x, y\rangle \in A \wedge y \in S\} \\ S_1 &= \{y : \langle x, y\rangle \notin A \wedge y \in S\} \\ \epsilon &= \frac{1}{g(n)} \end{aligned}$$

By Lemma 66 if $x \in L$ we have probability $\geq 1 - 2e^{-k\cdot\epsilon^2}$ that $|T_0| > |T_1| + k\epsilon/2$ and if $x \notin L$ we have probability $\geq 1 - 2e^{-k\cdot\epsilon^2}$ that $|T_0| + k\epsilon/2 < |T_1|$. Clearly, $k\epsilon/2 > c$ for all large inputs.

We know how to compare the number of DNA sequences in the two test tubes. By Theorem 45, we finish the proof of the theorem.

∎

6.5. Oracle Separating RP and BPP Hierarchies

We give some evidence that the molecular computer can improve the power of random computation by constructing the following oracle.

Theorem 70. *Let f, g be two nondecreasing functions from N to N. If $g(n) \neq f(n) \cdot n^{O(1)}$, then there is an oracle A such that $\mathrm{RP}^A[\frac{1}{g(n)}] \not\subseteq \mathrm{BPP}^A[\frac{1}{f(n)}]$.*

Proof: Let $M_1, M_2, \ldots,$ be an enumeration of all deterministic Turing machines. Each machine M_k runs in time $p_k(n) = n^k + k$. For any language A, define

$$L_A = \{x : \mathrm{Prob}[y : \langle x, y\rangle \in A \wedge y \in \Sigma^{|x|}] \geq \frac{1}{g(n)}\}.$$

Clearly, $L_A \in \mathrm{RP}[\frac{1}{g(n)}]$.

Lemma 71. *For any oracle B, $L \in \mathrm{BPP}^B[\frac{1}{f(n)}]$, there is a $p_k(n)$ time deterministic Turing machine M_k such that*

$$x \in L \implies \mathrm{Prob}[y : M_k \textit{ accepts } \langle x, y\rangle \textit{ and } y \in \Sigma^{p_k(|x|)}] \geq \frac{1}{2} + \frac{1}{g(n)}$$
$$x \notin L \implies \mathrm{Prob}[y : \textit{ accepts } \langle x, y\rangle \textit{ and } y \in \Sigma^{p_k(|x|)}] \leq \frac{1}{2} - \frac{1}{g(n)}.$$

Proof: Suppose $L \in \mathrm{RP}^B[\frac{1}{f(n)}]$ with the set $A \in \mathrm{P}^B$ and polynomial $p(n)$. Define

$$A' = \{\langle x, y'\rangle : y' = yz \text{ for some } z \bigwedge y \in \Sigma^{p(n)} \bigwedge \langle x, y\rangle \in A\}.$$

Clearly $A' \in \mathrm{P}^B$.

Clearly, there is a deterministic Turing machine M_k and polynomial $p_k(n)$ such that $A' = L(M_k^B)$ and $p_k(n) > p(n)$ for all n. It is easy to see that M_k satisfies the conditions of the lemma. ∎

In the following construction, we assume $|\langle x, y\rangle| \leq q_0(|x| + |y|)$ for a fixed polynomial.

Construction

Stage 0, $A_0 = \emptyset$.

Stage k, Let n_k be the least integer n such that $f(n)p_k(m) < g(n) - 1$

and none of the strings of length n_k is queried in any stage $k' < k$,

where $m = q_0(n + p_k(n))$.

Case 1. $M_k^{A_{k-1}}(\langle 0^{n_k}, y\rangle)$ accepts for more than $(\frac{1}{2} - \frac{1}{f(n)}) \cdot 2^{p_k(n_k)}$ ys in $\Sigma^{p_k(n_k)}$.

Let $A_k = A_{k-1}$.

Case 2. $M_k^{A_{k-1}}$ rejects $\langle 0^{n_k}, y\rangle$ for at most $(\frac{1}{2} - \frac{1}{f(n)}) \cdot 2^{p_k(n_k)}$ ys in $\Sigma^{p_k(n_k)}$.

Choose $\frac{2^{n_k}}{g(n_k)}$ elements B from Σ^{n_k} that are least queried by $M_k^{A_{k-1}}$ on $\langle 0^{n_k}, y\rangle$ with $y \in \Sigma^{p_k(n_k)}$. Let $A_k = A_{k-1} \cup B$.

The number of ys in $\Sigma^{p_k(n_k)}$ that $M_k^{A_{k-1}}$ queries B on input $\langle 0^{n_k}, y\rangle$ is at most

$$\frac{2^{p_k(n_k)} \cdot p_k(m)}{2^{n_k} - \frac{2^{n_k}}{g(n_k)}} \cdot \frac{2^{n_k}}{g(n_k)} = \frac{2^{p_k(n_k)} \cdot p_k(m)}{g(n_k) - 1} < \frac{1}{f(n)} \cdot 2^{p_k(n_k)}.$$

Hence,

$$\mathrm{Prob}[y : M_k^{A_t} \text{ accepts } \langle x, y\rangle \bigwedge y \in \Sigma^{p_k(n_k)}] < \frac{1}{2}.$$

Let $A = \bigcup_{k=0}^{\infty} A_k$.

End of Construction.

Assume $L_A \in \mathrm{BPP}^A[\frac{1}{f(n)}]$ via M_k^A and $p_k(n)$. Let's consider stage k of the construction.

If Case 1 holds, then $0^{n_k} \notin L_A$ and $M_k^A(x, y)$ accepts for more than $\frac{1}{f(n)} \cdot 2^{p_k(n_k)}$ ys in $\Sigma^{p_k(n_k)}$. This is a contradiction.

If Case 2 holds, then $x \in L_A$ and $\mathrm{Prob}[y : M_k^{A_t}$ accepts $\langle x, y\rangle \wedge y \in \Sigma^{p_k(n_k)}] < \frac{1}{2}$. This is also a contradiction.

∎

From the definition, RP (BPP) is a subset of $\mathrm{RP}[\frac{1}{f(n)}]$ ($\mathrm{BPP}[\frac{1}{f(n)}]$) for any super-polynomial function $f(n)$. The following corollary says that RP (BPP) is a strict subset of $\mathrm{RP}[\frac{1}{f(n)}]$ ($\mathrm{BPP}[\frac{1}{f(n)}]$) relative to some oracles.

Corollary 72. *Let $f(n)$ be a super-polynomial function.*

- *There is an oracle A such that* $\mathrm{RP}^A \neq \mathrm{RP}^A[\frac{1}{f(n)}]$.
- *There is an oracle A such that* $\mathrm{BPP}^A \neq \mathrm{BP}^A[\frac{1}{f(n)}]$.

Chapter 7.

A 2^n–Volume Molecular Algorithm for the Hamiltonian Path Problem

Adleman [1] solved the Hamiltonian path problem by operating on DNA sequences. He used exhaustive search for the solution of a 7–cities instance. For a general n–cities instance, exhaustive search involves $n!$ number of cases.

In [9, 3, 39], some effort is made to decrease the volume in molecular computation by transforming some good classical algorithms into molecular algorithms. Held and Karp [28] have an $n^{O(1)}2^n$ time dynamic-programming algorithm that uses exponential space. We cannot transform their dynamic programming algorithm into a molecular algorithm. The best known algorithm for the Hamiltonian path problem is Bax's algorithm [4], which uses $n^{O(1)}2^n$ steps and polynomial space. The techniques used in [9, 3, 39], cannot be applied in transforming Bax's algorithm [4] into a molecular algorithm. The main barrier is that DNA sequences have little ability of communication. In this paper we try to make DNA sequences have some communication ability and use practical operations. Our method is to let one DNA sequence randomly find another DNA sequence to combine together. We can use the information in both sequences to do computation. Our method transforms Bax's algorithm into a molecular algorithm. The number of DNA sequences used in our algorithm is 2^n, which is much less than $n!$.

7.1. The Hamiltonian Path Problem and Bax's Algorithm

In this section, we describe Bax's algorithm. In the next section, we show how to implement it in the molecular model.

Definition 73. Let $G = (V, E)$ be a graph with a set V of n vertices and a set E of edges.

- A Hamiltonian path of G is a path that goes through each vertex of G exactly once.
- $H_{n-1}(u, v)$ is the number of Hamiltonian paths from u to v.
- Let the adjacency matrix of G be A, *i.e.*, for any $u, v \in V$,

$$A(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

- For any $S \subseteq V$, $W_S(u, v)$ is the set of all paths of length $n - 1$ from u to v that have no vertex in set S.
- For any $S \subseteq V$,

$$A_S(u,v) = \begin{cases} A(u,v) & \text{if } u \notin S \text{ and } v \notin S \\ 0 & \text{otherwise} \end{cases}$$

By the inclusion and exclusion principal,

$$H_{n-1}(u,v) = \sum_{S \subseteq V} (-1)^{|S|} |W_S(u,v)|.$$

Furthermore, $|W_S(u, v)| = A_S^{n-1}(u, v)$. Since matrix multiplication can be done in $O(n^3)$ time, this implies the following lemma, which will be used in the implementation of the molecular algorithm. Bax's algorithm can be described as the function Hamiltonian.

Lemma 74. *There is a polynomial size circuit C_B such that for any graph $G = (V, E)$ and any $S \subseteq V$, it outputs $(-1)^{|S|}|W_S(u, v)|$.*

In the following sections, B_S denotes as $(-1)^{|S|}|W_S(u,v)|$.

function Hamiltonian(G,u,v)

for each $S \subseteq V$, $B_S = (-1)^{|V|} A_S^{n-1}(u,v)$;

return$(\sum_{S\subseteq V} B_S)$;

7.2. Molecular Algorithm

7.2.1. 2^n Robbers Problem

In a night 2^n robbers independently enter a village at the same time. Suppose robber i has treasures B_i for $i = 1, \ldots, 2^n$. If any two of them meet, one of them will kill the other and take his treasures. Within every t_0 time units each living robber meets another one, and kills him or is killed by him. We assume no three robbers meet together at the same time. Eventually there is only one robber left. He has all of the treasures $\sum_{i\leq m} B_i$. How long does it take until there is only one robber left?

Since during every t_0 time units, half of robbers will be killed, it is easy to see that the total time is $t_0 \cdot n$. If we replace the condition "Within every t_0 time units each living robber meets another one, and kills the other or is killed" by "Within every time t_0 at least 40 percent of living robbers meets another one, and kills the other or is killed". It is also easy to see that after $O(t_0 \cdot n)$ time, there is only one robber left.

This story is the idea of our molecular algorithm. Each robber is represented by a DNA sequence. Each treasure is B_S. Two robbers meet via annealing. The one who is killed is deleted.

7.2.2. Some New Molecular Operations

Definition 75. A *partial double DNA strand* is a DNA strand $x(\updownarrow s)y$ for some x, s, y.

Melting(T)

if $x(\updownarrow s)y \in T \Rightarrow x(\updownarrow s)y \notin T$, $xs \in T$ and $y^\wedge\overline{s}^\wedge \in T$.

Note: $x(\updownarrow s)y$ is considered as the combination of xs with $y^\wedge\overline{s}^\wedge$ via s and $\overline{s}^\wedge$. This operation separates the partial double sequence by heating the solution.

α-**Partition**(T, T_1, T_2)

if $|T| > 1$, then T is partitioned into T_1 and T_2 with $|T_1| > a|T|$ and $|T_2| > a|T|$. Note: we require $0 \leq \alpha \leq 0.5$. Throughout this paper α is set to $\frac{1}{4}$.

Test(T, k)

False if T contains more than k elements; True, Otherwise.

In this paper we only use it for $k = 1$. We need Test if a test tube contains more than one DNA sequences and do not wish to destroy the solution. In order to keep the solution we amplify the solution to another tube T^*. We use the techniques in the last chapter to compare T^* with T', which contains one DNA sequence. Test$(T, 1)$ can be implemented by α-Partition(T, T_1, T_2); Test(T_1) or Test(T_2);

Append$^*(T, a)$

Before the operation:

T contains all strings of the form $x(\updownarrow s)y$.

After the operation:

T contains all strings $ax(\updownarrow s)ya$ for every $x(\updownarrow s)y$ that was in T before the operation.

In DNA synthesis, new characters are usually appended to the end(3') of DNA sequences. In [11, 21], a method is described that makes it possible to append new characters at the other end($5'$). Our operation Append* is based on their principles.

7.2.3. Brief Description of the Algorithm

Definition 76. Fix the vertices $V = \{v_1, \ldots, v_n\}$ of graph G. For $S \subseteq V$, $k(S) = a_1 \cdots a_n$, where $a_i = \begin{cases} 1 & \text{if } v_i \in S \\ 0 & \text{otherwise} \end{cases}$

We first produce 2^n DNA sequences that encode $k(S)$ for all $S \subseteq V$. Since DNA operations can simulate the circuit, we can get 2^n DNA sequences such that each one has appended to it the information B_S.

The next step is to add all those B_S in the 2^n different DNA sequences together. Suppose all the 2^n sequences are in test tube T_0. Partition T_0 evenly into T_1 and T_2. Append string s to all sequences in T_1 and append $\overline{s}^\wedge$, the reverse of complement of the s, to all sequences in T_2. Merge T_1 and T_2 into T_0. The strings between T_1 and T_2 will link together in T_3 via s and $\overline{s}^\wedge$. Separate all partially double sequences in T_3 and put them into T^*. For two linked strings, add their values, and append the sum. Heat the solution T^* to divide all double sequences into single sequences. Delete those single sequences in T^* from T_2 and merge all other single sequences. So, the volume is decreased by almost half. Repeat this until there is only one sequence left that encodes $H_{n-1}(u, v)$.

Generate $T_0 = \{k(S) : S \subseteq V\}$;
Generate $T_0 = \{k(S)\#B_S : S \subseteq V\}$;
Repeat
  Partition T into T_1, T_2 evenly;
  Produce a new DNA sequence u and append f_1 to T_1;
  Produce a new DNA sequence v and append f_2 to T_2;
  Produce a new DNA sequence s;
  Append s to T_1;
  Append $\overline{s}^\wedge$ to T_2;
  Anneal T_1 and T_2 into T_0;
  Separate all partial double sequences and put them into T^*;
  For all partial double sequences, append sum.
  Melt T^* so that all partial double sequences split into two single sequences.
  Separate all sequences from T^* with information v;
  Merge T and T^* into T_0;
until T has only one string left.

7.2.4. Formal Description of the Algorithm

Definition 77.

- For $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$, $\langle x, y \rangle = 0001x_{11}x_2 \cdots 1x_n1001y_{11}y_2 \cdots y_m1000$.

- $\langle x_1, \ldots, x_m \rangle = \langle \langle x_1, \ldots, x_{m-1} \rangle, x_m \rangle$.

- For a tuple $K = \langle k_1, \ldots, k_m \rangle$, $K(i) = k_i$ for $i = 1, \ldots, m$.

- For a DNA sequence d and a tuple $K = \langle k_1, \ldots, k_m \rangle$. The interpretation of d at K, denoted by $\text{int}(d, K) = x_1 \cdots x_m$, where

$$x_i = \begin{cases} 0 & \text{if } \langle k_i, 0 \rangle \text{ is a substring of } d, \\ \\ 1 & \text{Otherwise} \end{cases}$$

- For a test tube T and a DNA sequence $d \in T$ and a set of operations A, $A(d)$ is the sequence that is transformed from d by A.

- For $\langle x, y_n \cdots y_0 \rangle \in \{0, 1\} \times \Sigma^*$,

$$\beta\langle x, y \rangle = \begin{cases} \sum_{i=0}^{n} y_i 2^i & \text{if } x = 1 \\ \\ -\sum_{i=0}^{n} y_i 2^i & \text{if } x = 0 \end{cases}$$

We fix an integers w_0 such that for any two DNA sequences d and d', ds can be linked to $d\overline{s}^{\wedge}$ for any s of length w_0 via $s, \overline{s}^{\wedge}$. Let H be a hash function from N to $\{A, T, G, C\}^{w_0}$ with the following properties:

- $H(n) \neq B_i(a)$ for any n, i and $a \in \{0, 1\}$; and

- It is hard to find two $i \neq j$ with $H(i) = H(j)$.

Lemma 78. *There is a polynomial size circuit C_{Sum} such that for two $\langle x, y \rangle$ and $\langle x', y' \rangle$ in $\{0, 1\} \times \Sigma^*$, it outputs $\langle u, v \rangle \in \{0, 1\} \times \Sigma^*$ with $\beta(\langle u, v \rangle) = \beta(\langle x, y \rangle) + \beta(\langle x', y' \rangle)$.*

Proof: Addition,encoding, $\langle \cdot, \cdot \rangle$, and β are all polynomial time computable. So, we have polynomial size circuit C_{Sum} to compute $\beta(\langle x, y \rangle) + \beta(\langle x', y' \rangle)$. ∎

Lemma 79. *Let C be a circuit with output gates $g_{i_1}, \ldots, g_{i_m}$; $K = \langle k_1, \ldots, k_m \rangle$ be a tuple; r be an integer: and T_0 be a test tube of DNA sequences. Assume that for any $d \in T_0$, it does not contain any subsequence $\langle r, u, v \rangle$ for any u, v. There is a $O(\text{size}(C))$ time molecular algorithm A such that for any DNA sequence d in T_0, $C(\text{int}(d, K)) = y_1 \cdots y_m$ iff $\langle r, i_j, y_i \rangle \subseteq A(d)$ for $1 \leq j \leq m$.*

Proof: The proof is similar to that of Lemma 9. We use more complicated encoding techniques. We are going to design a function Compute(C, K, r, T_0), which evaluates $C(\text{int}(d, K))$ for each DNA sequence d in test tube T_0. The result is appended to d.

Let C's input gates be $g_1, \ldots, g_n$, the internal gates are $g_{n+1}, \ldots, g_m$ in topological order. For each i, let g_i compute $f_i(g_{j(i)}, g_{t(i)})$ where $j(i) < i, t(i) < i$ and f_i is binary function.

(1) function Compute(C, K, r, T_0)
(2) for $i = n + 1$ to m do
(3) if $g_{j(i)}$ is input gate then Sep$(T_0, \langle K(j(i)), 0\rangle, T_1, T_2)$
(4) else Sep$(T_0, \langle r, j(i), 0\rangle, T_1, T_2)$;
(5) if $g_{t(i)}$ is input gate then Sep$(T_1, \langle K(t(i)), 0\rangle, T_0, T_3)$
(6) else Sep$(T_1, \langle r, t(i), 0\rangle, T_0, T_3)$;
(7) Append$(\langle r, i, f_i(0, 0)\rangle, T_0)$;
(8) Append$(\langle r, i, f_i(0, 1)\rangle, T_3)$;
(9) Merge(T_0, T_3, T_1);
(10) if $g_{t(i)}$ is input gate then Sep$(T_2, \langle K(t(i)), 0\rangle, T_0, T_3)$
(11) else Sep$(T_2, \langle r, t(i), 0\rangle, T_0, T_3)$;
(12) Append$(\langle r, i, f_i(1, 0)\rangle, T_0)$;
(13) Append$(\langle r, i, f_i(1, 1)\rangle, T_3)$;
(14) Merge(T_0, T_3, T_2);
(15) Merge(T_1, T_2, T_0);
(16) return(T_0);

The correctness is similar to that of Lemma 9. The i-th input of C is 0 in $d \in T_0$ if $\langle K(i), 0\rangle$ is the subsequence of d. Through the computation, if the value of gate g_i is a_i, $\langle r, i, a_i\rangle$ will be appended to the corresponding sequence for $i = n + 1, \ldots, m$.

Let's check each cycle of the loop for $n + 1 \leq i \leq m$. Right after (6), T_0 contains all sequences whose $g_{j(i)}$ value and $g_{k(i)}$ value are $0, 0$, and T_3 contains all sequences whose $g_{j(i)}$ value and $g_{k(i)}$ value are $0, 1$ respectively. At (7), we append $f_i(0, 0)$ to all sequences in T_0 as all of them have the same value $f_i(0, 0)$ at gate g_i. Similarly we append $f_i(0, 1)$ to all sequences in T_3 at (8).

Right after (11), T_0 contains all sequences whose $g_{j(i)}$ value and $g_{k(i)}$ value are $1, 0$, and T_3 contains all sequences whose $g_{j(i)}$ value and $g_{k(i)}$ value are $1, 1$ respectively. At (12), we append $f_i(1, 0)$ to all sequences in T_0 as all of them have the same value $f_i(1, 0)$ at gate g_i. Similarly we append $f_i(1, 1)$ to all sequences in T_3 at (13).

At completion, T_1 contains all strings that satisfy C and T_2 contains all strings that do not satisfy C.

The time follows from the algorithm directly. ∎

Lemma 80. *Let C be a circuit with output gates $g_{i_1}, \ldots, g_{i_m}$; $K = \langle k_1, \ldots, k_m \rangle$ be a tuple; r be an integer; and T be a test tube of partial double DNA sequences. Assume that for any $d \in T$, it does not contain any subsequence $\langle r, u, v \rangle$ for any u, v. There is an $O(\text{size}(C))$ time molecular algorithm A such that for any $d = x(\updownarrow s)y$ in T, $A(d) = zx(\updownarrow s)yz^*$ for some z and $C(\text{int}(d, K)) = y_1 \cdots y_m$ iff $\langle r, i_j, y_i \rangle \subseteq A(d)$ for $1 \le j \le m$.*

Proof: The proof is similar to that of Lemma 79. We use Compute* to replace Compute. The only difference between Compute* and Compute is that Append is replaced by Append*. ∎

Lemma 81. *Let T_0 be a test tube of DNA sequences; and i be an integer; and $K = \langle k_1, \ldots, k_m \rangle$ be a tuple. Assume that for any $d \in T$, d does not contain any subsequence $\langle i, u, v \rangle$ for any u, v. There is an $O(m)$ time molecular algorithm A to output another test tube T' such that for each $d \in T$, $\text{int}(d, K) = \text{int}(A(d), \langle \langle i, 1 \rangle, \ldots, \langle i, m \rangle \rangle)$.*

Proof:

(1) procedure Copy(i, K, T);

(2) for $j = 1$ to m do

(3) Sep$(T, \langle K(j), 0 \rangle, T_0, T_1)$;

(4) Append$(T_0, \langle i, j, 0 \rangle)$;

(5) Append$(T_1, \langle i, j, 1 \rangle)$;

(6) Merge(T_0, T_1, T);

Let's consider the ith cycle of the loop. Right after (3), T_0 contains all of the sequences d such that $\langle K(j), 0 \rangle$ is a subsequence of d. This means the $i$bit of $\mathrm{int}(d, K)$ is 0 for all d in T_0. Similarly the $i$bit of $\mathrm{int}(d, K)$ is 1 for all d in T_1. In (4) and (5), T_0, T_1 are appended $\langle i, j, 0\rangle, \langle i, j, 1\rangle$ respectively. We have $int(d, \langle K(j)\rangle) = int(A(d), \langle\langle i, j\rangle\rangle)$ for all d. This holds for $j = 1, \ldots, m$. We have that for each $d \in T$ (at the beginning), $\mathrm{int}(d, K) = \mathrm{int}(A(d), \langle\langle i, 1\rangle, \ldots, \langle i, m\rangle\rangle)$. ▮

Lemma 82. *Let T_0 be a test tube of DNA sequences; r_0 be an integer; and $K = \langle k_1, \ldots, k_m\rangle$ be a tuple. Assume that for any d in T_0, $\mathrm{int}(d, K) \in \{0, 1\} \times \Sigma^*$ and d does not contain any subsequence $\langle r, u, v\rangle$ for any u, v and any $r \geq r_0$. There is a $(l(T_0) + \log(|T_0|))^{O(1)}$ time molecular algorithm to output a single DNA sequence d' and a tuple K' such that $\mathrm{int}(d', K') \in \{0, 1\} \times \Sigma^*$ and*

$$\beta(\mathrm{int}(d', K')) = \sum_{d \in T} \beta(\mathrm{int}(d, K)).$$

Proof:

function $\mathrm{Add}(T_0, r_0, K)$;
    $i = r_0$;
    Repeat
        α-partition(T_0, T_1, T_2);
        Copy$(i + 1, K, T_1)$;
        Copy$(i + 2, K, T_2)$;
        $u_i = H(i); v_i = H(i + 1); s_i = H(i + 2)$;
        Append(T_1, u_i);
        Append(T_1, s_i);
        Append(T_2, v_i);
        Append$(T_2, \overline{s}_i^{\wedge})$;
        Anneal(T_1, T_2, T_3);
        Sep(T_3, u_i, T_4, T_5);
        Sep(T_4, v_i, T_6, T_7);
        m is the number of elements in K;
        $K_1 = \langle\langle i + 1, 1\rangle, \ldots, \langle i + 1, m\rangle, \langle i + 2, 1\rangle, \ldots, \langle i + 2, m\rangle\rangle$;
        Compute*$(C_{\mathrm{Sum}}, K_1, T_6)$;
        Let K be the set of all output gates of C_{Sum} ;
        Melting(T_6);

(20) Separate(T_6, u_i, T_8, T_9);
(21) Merge(T_8, T_7, T_{10})
(22) Merge(T_{10}, T_5, T_0);
(23) $i = i + 3$;
(24) until Test($T_0, 1$);
(25) return(T_0, K);

Step (4) is to partition the solution into two balance parts. Steps (5)-(6) copy the old result. In (7)-(11), T_1 are appended $u_i s_i$ and T_2 are appended $v_i \overline{s}_i^{\wedge}$. In (12), T_1 and T_2 are annealed. The strings in T_1 and T_2 combine together via $s_i, \overline{s}_i^{\wedge}$. After (13)-(14), T_6 contains all partial double strings in T_3, T_5 contains all single strings in T_3 with v_i, and T_7 contains all single strings in T_3 with u_i. In (15)-(18), compute the sum of the values at two combined strings. In (19), all partial double strings are changed into single strings. In (20)-(22), we delete all of the single sequences that were a part of partial double sequences and contain information v_i. All of the other single sequences are put into tube T_0. The loop stops when there is one string left. This string contains the result we need.

During the loop of Add, each cycle deletes $\frac{\alpha}{2}$ fraction DNA sequences. The number of cycles in the loop of Add is m, where m is the least number such that $|T_0| \cdot (\frac{3}{4})^m \leq 1$. Hence $m = O(\log v)$. On the other hand, C_{Sum} is a polynomial size circuit. The time for Compute*(C_{Sum}, K_1, T_6) is bounded by $|l(T_0)|^{O(1)}$. ▮

Theorem 83. *Let $g : \mathsf{N} \to \mathsf{N}$ be a polynomial time computable function. There is a molecular algorithm to compute $\sum_{i=1}^{v} g(i)$ in time $(\log v)^{O(1)}$ and volume v.*

Proof: Let C be a polynomial size circuit for $g(i)$.

The molecular algorithm is described as follows.

(1) Sum(C, v);
(2) Init(T_0, v);
(3) Compute($0, C, T_0$);
(4) $K = \langle k_1, \ldots, k_m \rangle$, where $g_{k_1}, \ldots, g_{k_m}$ are the output gates of C.
(5) Add($T_0, 1, K$);

From (3), each DNA sequence d is appended $g(i)(1 \leq i \leq v)$. By Lemma 82, T_0 contains a single DNA sequence d' such that $\text{Init}(d', K') = \sum_{i=1}^{m} g(i)$ after (5).

After the Init operation, we know that the volume of this algorithm is v. The time follows from the time in previous lemmas. ∎

Corollary 84. *There is a Molecular algorithm for Hamiltonian path problem that uses volume 2^n and polynomial steps.*

7.3. Application to Integration

In last section we design an algorithm to compute $\sum_{i=1}^{v} g(i)$. This algorithm can be applied in computing integration. For a function f, the integration of f over area A is

$$\lim_{\max\{v(A_1),\ldots,v(A_k)\}\to 0} \sum_{i=1}^{k} f(x_i)v(A_i),$$

where area A is partitioned into small areas $A_1, \ldots, A_k$, $v(A_i)$ is the volume (measure) of A_i, and each $x_i \in A_i(i = 1, \ldots, k)$. If A is an area in high dimensional space, molecular algorithm is theoretically efficient. An interesting special case of integration is volume computation.

Chapter 8.

Conclusions and Open Problems

In Chapter 3, we show that a large class of recursive algorithms can be transformed into molecular algorithms. An interesting problem is to find a larger class of algorithms in the classical computation model that can be transformed into molecular algorithms. In Chapter 7, we give one example. Dynamic programming algorithms cannot be transformed into molecular algorithms by our method.

In Chapter 4, we show that MOL($s(n)$) = MOL-A($s(n)$) iff NPinit($s(n)$) = NPpaths($s(n)$). Obviously, if P = NP then NPinit($s(n)$) = NPpaths($s(n)$). How does NPinit($s(n)$) $\stackrel{?}{=}$ NPpaths($s(n)$) relate to other questions in complexity theory? Is NPpaths($s(n)$) $\subseteq$ NPinit($s'(n)$) for some $s'(n)$ that is not much larger than $s(n)$, say $s'(n) = O(s(n))$ or $s'(n) = s(n)^{O(1)}$? A large class of recursive algorithms can be converted to NPinit($s(n)$) algorithms. It would be interesting to look for a larger class of algorithm thats could be converted to NPpaths($s(n)$) algorithms.

Our simulation of MOL$'$($s(n)$) by NPinit($s(n)$) maps $t(n)$-time molecular algorithms to $t(n)^4$-time RAM algorithms (we have not closely examined the Turing machine implementation of these algorithms). The simulation of NPinit($s(n)$) by MOL($s(n)$) maps $t(n)$-time Turing machine algorithms to $t(n) \log t(n)$-time molecular algorithms, because of the overhead for converting a Turing machine to a circuit. Can the simulations be performed more efficiently, say at the cost of a constant factor in running time?

In Chapter 5 we found a trade-off between the error-ratio and volume of the molecular algorithm for an approximation problem. It gives us more flexibility to use molecular algorithm in dealing with approximation problems. It would be interesting to find more subexponential volume algorithms for more natural problems.

In Chapter 6, we show how to deal with PP with molecular computation. In our algorithm we try to use practical operations. An interesting topic for further research would be to find larger complexity classes that can be dealt with using only practical operations. We also showed that a molecular computer may be more powerful than a conventional computer. An interesting problem is to find a natural problem that is in $\mathrm{BPP}[\frac{1}{g(n)}]$, but that is unlikely to be in BPP. Another interesting problem is to identify more natural algorithms to be in the class P^{PP} since we showed that this class can be handled by a molecular computer.

In this thesis, we ignore the analysis of the number of operations; which is also an important complexity measure in molecular computation. A step of a molecular computation costs usually more than 20 minutes. A polynomial time bound is still far from practical possibility, but it is a first good step toward it.

The practical implementation of molecular algorithm depends on many aspects of the algorithm such as volume, time and the length of the DNA strands. A good molecular algorithm should only use practical biological operations and small volume and small number of operations.

Bibliography

[1] L. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, Nov. 1994.

[2] L. Adleman. On constructing a molecular computer. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*, pages 1–21. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[3] E. Bach, A. Condon, E. Glaser, and C. Tanguay. DNA models and algorithms for NP-complete problems. In *Proceedings of the 11th Annual Conference on Structure in Complexity Theory*, pages 290–299, 1996.

[4] E. T. Bax. Inclusinon and exclusion algorithm for the hamiltonian path problem. *Information Processing Letters*, 47:203–207, 1993.

[5] D. Beaver. Molecular computation. Technical report, Penn State University, 1995.

[6] D. Beaver. A universal molecular computer. CSE 95-001, Penn State University, 1995.

[7] R. Beigel. Maximum independent set algorithms. Manuscript, 1996.

[8] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 444–452, 1995.

[9] R. Beigel and B. Fu. Molecular computing, bounded nondeterminism, and efficient recursion. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, pages 816–826, 1997. LNCS 1256.

[10] R. Beigel and J. Goldsmith. Downward separation fails catastrophically for limited nondeterminism classes. In *Proceedings of the 9th Annual Conference on Structure in Complexity Theory*, pages 134–138, 1994.

[11] G. M. Blackburn and M. J. Gait. *Nucleic Acids in Chemistry and Biology.* IRL Press, 1990.

[12] D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[13] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. *Discrete Applied Math.*, 71:79–94, 1996.

[14] D. Boneh and R. Lipton. Making DNA computer error resistent. In *Proceedings of Second Annual DIMACS Workshop on DNA Computation*, 1996.

[15] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SICOMP*, 22:560–572, 1993.

[16] J. D. C. Àlvarez and J. Torán. Complexity classes with complete problems between P and NP-complete. In *Foundations of Computation Theory*, pages 13–24. Springer-Verlag, 1989. LNCS 380.

[17] W. Cai, A. Condon, R. Corn, Z. F. Elton Glaser, T. Frutos, Z. Guo, M. Lagally, Q. Liu, L. Smith, and A. Thiel. The power of surface-based DNA computation. Manuscript, 1996.

[18] J. Chen and D. H. Wood. A new DNA separation techniques with low error rate. In *Proceedings of Third Annual DIMACS Workshop on DNA Computation*, pages 43–56, 1997.

[19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the -65th Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[20] J. Díaz and J. Torán. Classes of bounded nondeterminism. *MST*, 23:21–32, 1990.

[21] F. Eckstein. *Oligonucleotides and Anologues.* Oxford University Press, 1991.

[22] R. Feynman. *Minaturization.* Reinhold, 1961.

[23] B. Fu and R. Beigel. A comparison of resource-bounded molecular computation models. In *Proceedings of 5th Israeli Symposium on Theory of Computing and Systems*, pages 6–11, 1997.

[24] B. Fu and R. Beigel. Molecular approximation algorithm for np-optimization problems. In *Proceedings of Third Annual DIMACS Workshop on DNA Computation (June 23-25, 1997, Philadephia)*, pages 93–101, 1997.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[26] J. Goldsmith, M. Levy, and M. Mundhenk. Limited nondeterminism. *SIGACT News*, pages 20–29, June 1996.

[27] J. Hartmanis. On the weight of computations. *Bulletin of the European Association for Theoretical Computer Science*, 55:136–138, 1995.

[28] M. Held and R. M. Karp. A dynamic programming approach to sequencing problem. *J. Soc. Indust. App. Math.*, 10:196–210, 1962.

[29] L. Hemachandra and S. Jha. Defying upward and downward separation. *Inf. & Comp.*, 121:1–13, 1995.

[30] D. S. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *J. ACM*, 32(1):130–136, 1985.

[31] L. Kari. DNA computing: Arrival of biological mathematics. *he Mathematical Intelligencer*, 19(2):9–22, 1997.

[32] R. M. Karp, C. Kenyon, and O. Waarts. Error-resilient dna computation. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 458–467, 1997.

[33] C. M. R. Kintala. *Computations with a restricted number of nondeterministic steps*. PhD thesis, Pennsylvania State University, University Park, PA, 1977.

[34] C. M. R. Kintala and P. C. Fischer. Computations with a restricted number of nondeterministic steps. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 178–185, 1977.

[35] C. M. R. Kintala and P. C. Fischer. Refining nondeterminism in relativized polynomial-time bounded computations. *SICOMP*, 9(1):46–53, Feb. 1980.

[36] R. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, Apr. 1995.

[37] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Math.*, 10:287–295, 1985.

[38] N. Morimoto, M. Arita, and A. Suyama. Solid phase DNA solution to the hamiltonian path problem. In *Proceedings of Third Annual DIMACS Workshop on DNA Computation*, pages 93–101, 1997.

[39] M. Ogihara. Breadth first search 3SAT algorithms for DNA computers. Technical Report 629, University of Rochester, July 1996.

[40] M. Ogihara and A. Ray. DNA-based parallel computation by counting. In *Proceedings of Third Annual DIMACS Workshop on DNA Computation, pp. 265-274.*, 1997.

[41] C. H. Papadimitriou and M. Yannakakis. On limited nondeterminism and the complexity of the V–C dimension. In *Proceedings of the 8th Annual Conference on Structure in Complexity Theory*, pages 12–18, 1993.

[42] N. Pippenger and M. Fischer. Relations among complexity measures. *J. ACM*, 26, 1979.

[43] D. Pollard. *Convergence of Stochastic Process*. Springer Verlag, 1984.

[44] J. Reif. Parallel molecular computation: Models and simulations. In *Proceedings he Seventh Annual ACM Symposium on Parallel Algorithms and Archi tectures*, pages 213–223, 1995.

[45] J. Robson. Algorithms for maximum independent sets. *J. Algorithms*, 7:425–440, 1986.

[46] D. Roos and K. Wagner. On the power of dna-computing. *Information and Computation*, 131:95–109, 1996. This version does not contain their QBF algorithm.

[47] P. Rothemund. A DNA and restriction enzyme implementation of Turing machines. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*, pages 75–119. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[48] S. Roweis, E. Winfree, , R. Burgoyne, N. V. Chelyapov, M. F.Goodman, P. W. Rothemund, and L. M. Adleman. A sticker based model for DNA computation. In *Proceedings of the Second Annual Meeting on DNA Based Computers*, 1996.

[49] J. Sambrook, E. Fritsch, and T. Maniatis. *Molecular Cloning.* Cold Spring Harbor, NY, 1989.

[50] L. Sanchis. Constructing language instances based on partial information. *International Jour. Found. Comp. Sci.*, 5(2):209–229, 1994.

[51] I. Schiermeyer. Pure literal lookahead: An $O(1,497^n)$ 3-satisfiability algorithm. Manuscript, August 14, 1996.

[52] C. P. Schnorr. Optimal algorithms for self-reducible problems. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 322–337, 1976.

[53] A. L. Selman. Natural self-reducible sets. *SICOMP*, 17:989–996, 1988.

[54] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[55] W. Smith and A. Schweitzer. DNA computers in vitro and vivo. Technical report, NEC, 1995.

[56] R. Tarjan. Finding a maximum clique. Technical Report 72-123, Cornell University, 1972.

[57] S. Toda. PP is as hard as the polynomial-time hierarchy. *SICOMP*, 20(5):865–877, 1991.

[58] J. Watson, M. Gilman, J. Witkowski, and M. Zoller. *Recombinant DNA (2nd edition).* Scientific American Books, W.H.Freeman and Co., 1992.

[59] J. Watson, N. Hoplins, J. Roberts, and et. al. *Molecular Biology of the Gene.* Benjamin/Cummmings Menlo Park CA, 1987.

[60] E. Winfree. On the computational power of DNA annealing and ligation. http://dope.caltech.edu/winfree/DNA.html.

[61] E. Winfree. Complexity of restricted and unrestricted models of molecular computation. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*, pages 187–198. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[62] E. Winfree, X. Yang, and N. Seeman. Universal computation via self-asembly of DNA:some theory and experiments. In *Proceedings of Second Annual DIMACS Workshop on DNA Computation*, pages 172–190, 1996.