An $\tilde{O}(2^n)$ Volume Molecular Algorithm for Hamiltonian Path

submitted to DNA '98

Bin Fu* Richard Beigel† Fang Xiao Zhou‡

Abstract

We design volume-efficient molecular algorithms for all problems in #P, using only reasonable biological operations. In particular, we give a polynomial-time $O(2^n n^2 \log^2 n)$-volume algorithm to compute the number of Hamiltonian paths in an n-node graph. This improves Adleman's celebrated $n!$-volume algorithm for finding a single Hamiltonian path.

1. Introduction

Molecular computation was first proposed by Feynman [10], 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.

DNA is the storage medium for genetic information. It is composed of units called nucleotides, distinguished by the chemical group (base) attached to them. The four bases are adenine, guanine, cytosine, and thymine, abbreviated as A, G, C, and T. Single nucleotides are linked end-to-end to form DNA strands. Each DNA strand has two chemically distinguishable ends, called the $5'$ end and the $3'$ end respectively. Thus a DNA strand can be considered oriented; by convention a DNA strand is denoted by its sequence of bases (A, G, C, T), starting at the $5'$ end and finishing at the $3'$ end. Conversely, DNA strands can be used to encode strings over the alphabet $\{A, G, C, T\}$ or any other alphabet.

By convention, a DNA strand is denoted by the sequence of characters obtained by reading it in the $5' \to 3'$ direction. If x is a DNA strand, $\overline{x}$ denotes the Watson–Crick complement of x, and $\updownarrow x$ denotes the double strand that results when x and $\overline{x}$ anneal. For example if $x = 5'\mathrm{AGGCT}3'$, then $\overline{x} = 3'\mathrm{TCCGA}5'$ and $\updownarrow x$ is the double strand $\substack{5'\mathrm{AGGCT}3' \\ 3'\mathrm{TCCGA}5'}$

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

*Address: Dept. of Electrical Engineering and Computer Science, 19 Memorial Dr W Ste 2, Bethlehem PA 18015-3084. Email: bif3@eecs.lehigh.edu. Web: http://www.eecs.lehigh.edu/~bif3. Supported in part by the National Science Foundation under grants CCR-9796317 and CCR-9700417. Research performed in part as a student at Yale University while visiting the University of Maryland Institute for Advanced Computer Studies.

†Address: Dept. of Electrical Engineering and Computer Science, 19 Memorial Dr W Ste 2, Bethlehem PA 18015-3084. Email: beigel@eecs.lehigh.edu. Web: http://www.eecs.lehigh.edu/~beigel. Supported in part by the National Science Foundation under grants CCR-9796317 and CCR-9700417 and by NASA under grant NAG 52895. Research performed in part during a sabbatical visit from Yale University to the University of Maryland's Human–Computer Interaction Lab.

‡Address: Dept. of Molecular Biophysics and Biochemistry, Yale University, New Haven, CT 06520-8114. Email: fxzhou@pantheon.yale.edu.

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 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.

Following Adleman's groundbreaking experiment, Lipton [13] designed a simple molecular algorithm to solve the boolean satisfiability problem (SAT). The boolean satisfiability problem is to determine whether a boolean formula has a truth assignment that makes the formula true. 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. The number of operations used in a molecular computation is called *time*. In [3, 15, 7, 14], efforts were made in optimize the volume of DNA solution. Various papers [19, 6, 17, 8] apply molecular computing to solve standard computational problems.

Beaver [6] and Roos and Wagner [18] designed DNA algorithms to solve a PSPACE-complete problem, but they used an operation that requires every strand of DNA in a test tube to anneal with its exact counterpart. This is biologically impractical, because it is like requiring every DNA strand to find a needle in a haystack.

Given a boolean formula f, #SAT(f) is the number of satisfying assignments of f. #SAT plays a very important role in complexity theory. For example, the polynomial time hierarchy can be reduced to #SAT via polynomial time Turing reductions [20]. We design a molecular algorithm for #SAT that uses only practical operations. This greatly improves on what was previously known to be possible using practical molecular operations. We also design a molecular algorithm for computing the number of Hamiltonian paths that uses polynomial time and $O(2^n n^2 \log^2 n)$ volume.

2. DNA Computation Model

Laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [21, 22].

The computational models we define in this section were inspired by the work of [3, 18]. A test tube is a multiset of binary strings. We describe the allowable operations formally 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 size of the solution space (also called simply *volume*), which is the maximum number of strings in all test tubes at any time, counting multiplicities. Adleman [2] has speculated that molecular computation with a solution space of size 2^{70} (about 0.002 moles) might be possible. Recent papers [3, 15] attempt to optimize solution volume size for particular combinatorial problems.

2.1. Encoding

Let $x_1 \cdots x_n$ be an n-bit binary string. We choose a 30-base DNA strand B_0, and, for notational convenience define $B_0(0) = B_0(1) = B_0$. For each bit position $i \le n$ and each bit $b \in \{0, 1\}$, we choose a 30-base DNA sequence $B_i(b)$ such that, for all i, j, a, b, c, $B_i(a)$ has no long common subsequence with any $B_j(b)B_{j+1}(c)$ unless $(i, a) \in \{(j, b), (j+1, c)\}$.

Long common subsequences might cause DNA strands to be misclassified in Separation by subsequence. For in-depth discussion and a construction of the encoding, see Baum [4].

The string $x_1 \cdots x_n$ is represented by the DNA sequence

$$B_0 B_1(x_1) B_2(x_2) \cdots B_n(x_n).$$

Such DNA sequences are called *valid encodings*. (The initial sequence B_0 will be used in order to implement the Amplify operation with Polymerize, Append, and Merge, which will be defined shortly. It is also used for adding primer in polymerization)

2.2. Molecular Operations

Sep(T_1, s, T_2, T_3)

$T_2 :=$ the multiset of all DNA sequences in T_1 that contain the sub-sequence s anywhere;

$T_3 :=$ the multiset of all DNA sequences in T_1 that do not contain the sub-sequence s anywhere;

$T_1 := \emptyset$.

The following implementation method for the Sep operation is paraphrased from [1]: We first create many copies of $\overline{s}$. 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 s anneal to the $\overline{s}$ oligos anchored to the matrix. We wash away all strands that did not anneal, leaving behind only those strands that contain s, which can then retrieved from the matrix.

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

$T_2 :=$ the multiset 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 whose length is less than i;

$T_1 := \emptyset$.

Append(T, c)

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

Merge(T_1, T_2, T_3)

$T_3 := T_1 \cup T_2$;

$T_1 := \emptyset$;

$T_2 := \emptyset$.

Test(T)

If T is not empty then "true" else "false".

Polymerize(T) $T = \{x, \overline{x} : x \in T\}$.

The Polymerize operation can be implemented via the polymerase chain reaction (PCR). We only use Polymerize to implement Amplify for valid encodings. Therefore, the initial complementary segement of B_0 can be used as primer for polymerase chain reaction.

Amplify(T_1, T_2, T_3)
    $T_2 := T_1$;
    $T_3 := T_1$;
    $T_1 := \emptyset$.

Init(T, m)
    T contains n-bit binary representation of i ($i = 1, \ldots, m$) $a_0 \cdots a_{n-1}$ such that $\sum_{j=0}^{n-1} a_j 2^j = i$, where $n = \lfloor \log m \rfloor + 1$.

Sep12(T_1, T_2, T_3)
    $T_2 :=$ the multiset of all single strands in T_1
    $T_3 :=$ the multi set of all strands in T_1 that have any double region.
    $T_1 := \emptyset$.

We will discuss how to implement Sep12 when we use it in the function Compare(T_1, T_2). The Init operation is implemented via the Amplify, Append and Merge operations. We produce 2^n strands encoding all strings in $\{0,1\}^n$ as follows:

```
procedure Init(T, 2^n);
    T := {Λ};
    for i := 1 to n do
        Amplify(T, T_0, T_1);
        Append(T_0, 0);
        Append(T_1, 1);
        Merge(T_0, T_1, T);
```

In case m is not necessarily a power of 2, Init(T, m) is easily reduced to Init($T_k, 2^k$) for $0 \leq k \leq \log_2 m$.

Now let us see how to implement Amplify. We assume that the source tube T_1 contains only valid encodings, so each of them starts with B_0.

```
procedure Amplify(T_1, T_2, T_3)
    Polymerize(T_1);
    Sep(T_1, B_0, T_2, T_4);
    Polymerize(T_4);
    Sep(T_4, B_0, T_3, T_5);
```

3. Comparing Number of DNA Sequences in Two Tubes

Definition 1.

- D^ℓ is a string consisting of ℓ copies of the character D.
- Because A and T are complementary, the DNA strands A^ℓ and T^ℓ can combine anywhere, leaving some unpaired A's and T's on either end. We denote the set of possible double strands obtained in this way as follows:

$$\begin{bmatrix} A \\ T \end{bmatrix}^\ell = \left\{ A^{\ell_1} (\updownarrow A^{\ell_2}) T^{\ell_3} : \ell_2 > 0 \text{ and } \ell_1 + \ell_2 = \ell_2 + \ell_3 = \ell \right\}$$

- If $y = y_1 \cdots y_m$, $y^\wedge$ denotes the reverse of the string y, *i.e.*, $y_m \cdots y_1$.
- $|T|$ is the number of sequences in test tube T. $l(T)$ is the length of the longest DNA strand in T. $l'(T)$ is the length of the binary string encoded by that strand.
- Let $S(\cdot)$ denote the characteristic function of the set S, *i.e.*, $S(x) = 1$ if $x \in S$, 0 if $x \notin S$.

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.

3.1. Pair and Conquer

Suppose there are two groups of people. One group consists of a great number of men and the other group consists of a great number of women. We wish to determine which group is larger.

We merge the two groups. Let each man pair with a randomly selected woman. Remove all couples formed in that way. We can then determine if the number of men is greater than the number of women based on the individuals left over.

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 $\overline{s}^\wedge$ for the mutual attraction between them. 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.

3.2. A Simple Algorithm for Comparison

Hypothesis 1: There is a constant w_0 such that for 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, assume we have

$$\begin{aligned} T_1 &= \{x_1 s, \ldots, x_{m_1} s\} \\ T_2 &= \{y_1 \overline{s}^\wedge, \ldots, y_{m_2} \overline{s}^\wedge\} \\ T_3 &= \emptyset \\ |s| &= 30 \end{aligned}$$

After the operation, we will have $T_1 = T_2 = \emptyset$ and

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

We believe that 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 with 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 $O(\log |T_1|)$ if every time we have 20 percent of the sequences annealed.

function Compare(T_1, T_2)
    choose two DNA sequences $s_1 \neq s_2$ from $\{A, T, G, C\}^{30}$;
    Append(T_1, s_1);
    Append(T_2, s_2);
    Append(T_1, A^{w_0});
    Append(T_2, T^{w_0});
    Anneal(T_1, T_2, T_0);
    Sep12(T_0, T_0, T_3)
    if there is a single sequence from T_1 left in T_0 then return "greater"
    else if there a single sequence from T_2 left in T_0 then return "less"
    else return "equal".

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 each 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 $d \in \{\mathrm{A}, \mathrm{T}\}^*$ and let $s_1, s_2 \in \{\mathrm{G}, \mathrm{C}\}^*$). In the case that we know that all strings have the same length n, we may use $s_1 = B_{n+1}(0)$ and $s_2 = B_{n+1}(1)$. We will provide a method how to partition the DNA strings into several tubes according to their length (see Lemma 3).

There may be several ways to separate the partial double sequences and check if T_0 has any sequence from T_1 left. The following is one of them.

    Sep(T_0, s_1, T_3, T_4);
    Sep(T_3, s_2, T_5, T_6);
    if not Test(T_6) then return "greater"
    else if not Test(T_4) then return "less"
    else return "equal";

After the second Sep operation above, T_5 contains all partial double sequences $(d_1(\updownarrow A^{\ell_1})d_2)$ that are a combination of a sequence from T_1 and a sequence from T_2; T_4 contains all sequences from T_2 that cannot combine with any sequence in T_1; and T_6 contains all sequences from T_1 that cannot combine with any sequence in T_2.

Theorem 2. *Assuming Hypothesis 1, there is an $O(1)$-time* {Append, Separate, Anneal, Test}*-molecular algorithm to compare the number of DNA sequences in two test tubes.*

3.3. Revised Algorithm for Comparison

For the sake of greater practicality, we base our Anneal operation 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 $\ell = m(|x|, |y|)$, 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, assume we have

$$\begin{aligned} T_1 &= \{x_1 A^\ell, \ldots, x_{m_1} A^\ell\} \\ T_2 &= \{y_1 T^\ell, \ldots, y_{m_2} T^\ell\} \\ T_3 &= \emptyset \end{aligned}$$

where $|x_1| = \cdots = |x_{m_1}|$, $|y_1| = \cdots = |y_{m_2}|$, and $\ell = m(|x_1|, |y_1|)$. After the operation, we will have $T_1 = T_2 = \emptyset$, and

$$\begin{aligned} m_1 + c < m_2 \implies & T_3 = \{x_1 S_1 y_{i_1}^\wedge, \ldots, x_{m_1} S_{m_1} y_{i_{m_1}}^\wedge, y_{j_1} T^\ell, \ldots, y_{j_{m_2-m_1}} T^\ell\}, \\ & \text{where } \{i_1, \ldots, i_{m_1}\} \cup \{j_1, \ldots, j_{m_2-m_1}\} = \{1, \ldots, m_2\} \\ & \text{and } S_i \in \begin{bmatrix} A \\ T \end{bmatrix}^\ell mbox \text{ for } i = 1, \ldots, m_1. \\ m_1 > m_2 + c \implies & T_3 = \{x_{i_1} S_1 y_1^\wedge, \ldots, x_{i_{m_2}} S_{m_2} y_{i_{m_2}}^\wedge, x_{j_1} A^\ell, \ldots, x_{j_{m_1-m_2}} A^\ell\}, \\ & \text{where } \{i_1, \ldots, i_{m_2}\} \cup \{j_1, \ldots, j_{m_1-m_2}\} = \{1, \ldots, m_1\} \\ & \text{and } S_i \in \begin{bmatrix} A \\ T \end{bmatrix}^\ell mbox \text{ for } i = 1, \ldots, m_2. \\ |m_1 - m_2| \le c \implies & \text{No result.} \end{aligned}$$

Anneal$_c$ is more practical than Anneal. Since the number of difference in two tubes is at least c, it is easy for a sequences in the small tube to find another sequence in the large tube to anneal with after many annealed sequences are separated. We only use this operation in the case $|m_1 - m_2| > c$.

Lemma 3. *There is an $O(l(T))$-time* {Separate, Merge, Test}*-molecular algorithm that takes a test tube T containing only valid encodings and produces tests tubes $T_0, T_1, \ldots, T_k$ such that $k \le l'(T)$ and $T_i = T \cap \Sigma^i$ for $i = 0, 1, \ldots, k$.*

Proof:

```
procedure LengthPartition(T)
    i := 1;
    while not Test(T) do
        Separate(T, 0, i, T', T'');
        Separate(T'', 1, i, T''', T_{i-1});
        Merge(T', T''', T);
        i := i + 1;
```

It is easy to see that $T_j = T \cap \Sigma^j$. The algorithm also outputs the number i, which is the length of the longest string encoded by a DNA strand in T. ∎

Lemma 4. *There is an* $O(l(T))$*-time* {Append, Separate, Merge}*-molecular algorithm that takes a test tube* T *containing only valid encodings and produces a test tube* T' *such that*

- $|T'| = T$ *and*
- *all sequences in* T' *have the same length.*

Proof: By Lemma 3, we can produce k tubes of DNA strands $T_1, \ldots, T_k$ such that $T_i = T \cap \Sigma^i$. We continue as follows:

$T' := \emptyset$;
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 5. *There is an* $O(m(l(T_1), l(T_2)))$ {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 set s equal to$A^{m(l(T_1),l(T_2))}$ and replace Anneal by Anneal$_c$ in Compare. ∎

In the proof of Lemma 5, we could also consider using Separation by length to separate all of the partial double sequences. This may make it easier to implement.

Theorem 6 follows from Lemmas 3, 4, and 5.

Theorem 6. *There is an* $O(\max\{l(T_1), l(T_2), m(l(T_1), l(T_2))\})$*-time* {Append, Merge, Separate, Anneal$_c$, Test}*-molecular algorithm* Compare$_c$ *that, given two test tubes* T_1, T_2 *of DNA with* $||T_1| - |T_2|| > c$*, determines whether* $|T_1| > |T_2| + c$ *or* $|T_1| + c < |T_2|$.

The following lemma is due to Boneh et al [9]. This version is given by Beigel and Fu in [7].

Lemma 7. *Let* π *be a circuit with* m *gates. Given a tube* T_0*, a molecular algorithm using only the operations 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 sequences* z *from tube* T_0 *that satisfy* $\pi(z) = 1$ *and* T_2 *contains all sequences* z *from tube* T_0 *that satisfy* $\pi(z) = 0$.

4. Counting

In this section we design molecular algorithms to recognize languages in PP. The idea is to generate all satisfying assignments and all unsatisfying assignments in the two tubes. This can be done using the Init operation and Lemma 7. Then, using the techniques from Section 3, we determine which tube contains more sequences.

Definition 8.

- A family of circuits $\{C_n\}_{n=1}^{\infty}$ is P-uniform if there is a polynomial time computable function f such that $f(0^n) = C_n$ for all n.
- NPinit($v(n)$) consists of all languages recognized by NP machines that nondeterministically choose a number between 1 and $v(n)$ on inputs of size n and then behave deterministically.
- $L \in$ PPinit[$v(n)$] if there is an NPinit($v(n)$) 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[$d(n), v(n)$]$'$ if there is an NPinit($v(n)$) 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 $d(n)$.
- For a function $f : \Sigma^* \longrightarrow \mathsf{N}$, $f(x) \in$ #Pinit[$g(n)$] if $f(x)$ is equal to the number of accepting paths of some NPinit($g(n)$) NTM M on input x with length n.
- A PP-MOL[$v(n)$] computation is a sequence consisting of an Init($T_0, v(n)$) operation, followed by a polynomial-bounded number of Append, Separate, and Merge operations, and finally a Compare operation. The computation *accepts* if the output of the Compare operations is "greater".
- A family of molecular computations (one for each input) is P-uniform if there is a deterministic polynomial-time Turing machine that maps each input to the sequence of molecular operations to be performed.
- $L \in$ PP-MOL[$v(n)$] if there is a P-uniform family of PP-MOL[$v(n)$] computations that accepts L.
- A PP-MOL[$c, v(n)$]$'$ computation is a sequence consisting of an Init($T_0, v(n)$) operation, followed by a polynomial-bounded number of Append, Separate, and Merge operations, and finally a Compare$_c$ operation. The computation *accepts* if the output of the Compare$_c$ operations is "greater".
- $L \in$ PP-MOL[$c, v(n)$]$'$ if there is a P-uniform family of PP-MOL[$c, v(n)$]$'$ computations that accepts L.

It is well known that Boolean circuits can simulate Turing machine efficiently [16]. We have the following proposition.

Proposition 9. *For an* NPinit($v(n)$) *Turing machine M, there is a* P*-uniform family of polynomial size circuits $\{C_n(.,.)\}$ such that $C_{|x|}(x, i)$ is equal to the result of M's ith path on input x.*

Lemma 10.

i. $\mathrm{PPinit}[v(n)] \subseteq \mathrm{PP\text{-}MOL}[v(n)]$.

ii. $\mathrm{PPinit}[c, v(n)]' \subseteq \mathrm{PP\text{-}MOL}[c, v(n)]'$.

Proof: (i) Let $L \in \mathrm{PPinit}[v(n)]$. There is a P-uniform family of polynomial-size circuits $\{\pi_n(\cdot,\cdot)\}_{n=1}^{\infty}$ such that for input of length n,

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

Let x be an input of length n. By Lemma 7, we can get two tubes T_1 and T_2 such that

$$\begin{aligned} T_1 &= \{y : 1 \leq y \leq v(|x|) \text{ and } \pi_n(x,y) = 1\}, \text{ and} \\ T_2 &= \{y : 1 \leq y \leq v(|x|) \text{ and } \pi_n(x,y) = 0\}. \end{aligned}$$

By Theorem 2, we can determine if $x \in L$ via $\mathrm{Compare}(T_1, T_2)$.

(ii) This is similar to the proof of (i). The only differences are that

$$x \in L \text{ iff } |\{y : 1 \leq y \leq v(|x|) \text{ and } \pi_n(x,y) = 1\}| > |\{y : 1 \leq y \leq v(|x|) \text{ and } \pi_n(x,y) = 0\}| + c;$$

and $\mathrm{Compare}(T_1, T_2)$ is replaced by $\mathrm{Compare}_c(T_1, T_2)$. ∎

Lemma 11.

i. $\mathrm{PP\text{-}MOL}[v(n)] \subseteq \mathrm{PPinit}[v(n)]$.

ii. $\mathrm{PP\text{-}MOL}[c, v(n)]' \subseteq \mathrm{PPinit}[c, v(n)]'$.

Proof: (i) This is similar to the proof of [11, Theorem 1]. The first operation of a $\mathrm{PP\text{-}MOL}[v(n)]$ computation is $\mathrm{Init}(T_0, v(n))$, which produces $v(n)$ DNA sequences encoding $1, 2, \ldots, v(n)$. Construct an $\mathrm{NPinit}(v(n))$ Turing machine such that path i simulates the DNA sequence that encodes i. If the final operation of the $\mathrm{PP\text{-}MOL}[v(n)]$ computation is $\mathrm{Compare}(T', T'')$, all paths that simulate the DNA sequences in T' accept and all paths that simulate the DNA sequences in T'' reject.

(ii) Very similar to the proof of (i). ∎

Theorem 12.

i. $\mathrm{PPinit}[v(n)] = \mathrm{PP\text{-}MOL}[v(n)]$.

ii. $\mathrm{PPinit}[c, v(n)]' = \mathrm{PP\text{-}MOL}[c, v(n)]'$

Proof: Follows from Lemma 10 and Lemma 11. ∎

Lemma 13. $\mathrm{PPinit}[v(n)] \subseteq \mathrm{PPinit}[d(n), d(n)(2v(n) + 1)]'$.

Proof: Let M be an NP TM that accepts L. Let M' be an NP TM derived from M by splitting each accepting path into two accepting paths and each rejecting paths into two rejecting paths; and adding one more rejecting path. If $x \in L$ then M' has more accepting paths than rejecting paths; if $x \notin L$ then M' has fewer accepting paths than rejecting paths. Let M'' be a TM derived from M' by splitting each accepting path into $d(n)$ accepting paths and each rejecting path into $d(n)$ rejecting paths. Clearly, if M is an $\mathrm{NPinit}(v(n))$ TM, then M'' is $\mathrm{NPinit}(d(n)(2v(n) + 1))$ TM. ∎

Theorem 14. $\mathrm{PPinit}[v(n)] \subseteq \mathrm{PP\text{-}MOL}[c, c(2v(n)+1)]'$.

Proof: It follows from Lemma 13 and Theorem 12. ∎

Theorem 15.

i. For every $f \in \#\mathrm{Pinit}[g(n)]$, there is an {Init, Append, Merge, Separate, Anneal, Test} molecular algorithm for $f(n)$ that runs in $n^{O(1)} \log g(n)$ time and $4g(n)$ volume.

ii. For every $f \in \#\mathrm{Pinit}[g(n)]$, there is an {Init, Append, Merge, Separate, Anneal_c, Test} molecular algorithm for f that runs in $n^{O(1)\log g(n)}$ time and $O(g(n))$ volume.

Proof: (i) Let $f \in \#\mathrm{Pinit}[g(n)]$ via an $\mathrm{NPinit}(g(n))$ NTM M. Define $L = \{\langle x, k\rangle : k \le g(n) \wedge M$ has more than k accepting paths $\}$. $L \in \mathrm{PPinit}[4g(n)]$ via an $\mathrm{NPinit}(k(n))$ NTM M' which is constructed from M as follows. Split each accepting path of M into 2 accepting paths, split each rejecting path of M into one accepting path and one rejecting path, add $g(n)+k$ additional rejecting paths, and $g(n)-k$ additional accepting paths. Clearly, $\langle x, k\rangle \in L$ iff the number of accepting path of M' is larger than the number of rejecting path on input x. The total number of paths is $4g(n)$.

Function $f(x)$ can be computed in polynomial time via binary query to L, where each query is of the form $\langle x, k\rangle$ with $k \le g(n)$. The answer can be got via molecular computation by Theorem 12. The number of queries is bounded by $\log g(n)$.

The proof of (ii) is similar. We use Theorem 14 instead of Theorem 12. ∎

It is easy to see that $\#\mathrm{SAT} \in \#\mathrm{Pinit}[2^n]$, where n is the number of variables in a formula. The function for number of Hamiltonian paths is in $\#\mathrm{Pinit}[n!]$, where n is the number of vertices of the input graph. In the next section we will design a molecular algorithm with volume only $O(2^n n^2 \log^2 n)$.

5. An $O(2^n n^2 \log^2 n)$-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 $n!$ cases.

Held and Karp [12] give a $2^n n^{O(1)}$-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 [5], which uses $n^{O(1)}2^n$ steps and polynomial space. The techniques used in [7, 3, 15], cannot be applied in transforming Bax's algorithm [5] into a molecular algorithm. Our new method transforms Bax's algorithm into a molecular algorithm. The number of DNA sequences used in our algorithm is $O(2^n n^2 \log^2 n)$, which is much less than $n!$. In this section, we describe Bax's algorithm and show how to implement it in the molecular model.

Definition 16. 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.

- $H(u,v)$ is the number of Hamiltonian paths from u to v.
- Let A denote the adjacency matrix of G, *i.e.*, for $u,v \in V$, $A(u,v)=1$ if $(u,v) \in E$, 0 otherwise.
- For $S \subseteq V$, $W_S(u,v)$ is the set of all paths of length $n-1$ from u to v that have no vertex in set S.
- For $S \subseteq V$, let
$$A_S(u,v) = \begin{cases} A(u,v) & \text{if } u \notin S \text{ and } v \notin S \\ 0 & \text{otherwise} \end{cases}$$

By the inclusion and exclusion principal, $H(u,v) = \sum_{S\subseteq V} (-1)^{|S|}|W_S(u,v)|$. 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.

Lemma 17. *There is a polynomial time computable function $f(G,S,i)$ such that for graph $G=(V,E)$, $S \subseteq V$, and $i \le \log_2(|V|!)$, $f(G,S,i) = b_{S,i}$, where $(-1)^{|S|}|W_S(u,v)| = \sum_{1\le i\le \log(|V|!)} b_{S,i}2^i$ with $b_{S,i} \in \{-1,0,1\}$.*

Let $B_S = (-1)^{|S|}|W_S(u,v)|$. Each B_S is an integer between $-|V|!$ and $|V|!$. It can be expressed as $B_S = \sum_{i=1}^{l} b_{S,i}2^i$, where $l=\log(|V|!)$ and $b_{S,i} \in \{-1,0,1\}$. Since $B_S = (-1)^{|S|}|W_S(u,v)|$, $b_{S,i} \in \{-1,0\}$ if $|S|$ is odd, and $b_{S,i} \in \{0,1\}$ if $|S|$ is even. Define
$$D(G,i,c) = \begin{cases} \sum_{S\subseteq V, |S| \text{ is even}} b_{S,i} & \text{if } c=0 \\ \sum_{S\subseteq V, |S| \text{ is odd}} -b_{S,i} & \text{if } c=1 \end{cases}$$

The desired total is
$$\begin{aligned} \sum_{S\subseteq V} B_S &= \sum_{S\subseteq V}\sum_{0\le i\le l} 2^i b_{S,i} \\ &= \sum_{0\le i\le l}\sum_{S\subseteq V} 2^i b_{S,i} \\ &= \sum_{0\le i\le l} 2^i \sum_{S\subseteq V} b_{S,i} \\ &= \sum_{0\le i\le l} 2^i(D(G,i,0) - D(G,i,1)). \end{aligned}$$

Lemma 18. $D(G,i,c) \in \#\mathrm{Pinit}[2^{|V|-1}]$.

Proof: If $c=1$, we let each number $1 \le j \le 2^n$ encode a unique subset S_j of V with odd number of elements. We can compute S_j in polynomial time of $|j|$. Let M be an $\mathrm{NPinit}(2^{|V|-1})$ NTM such that M accepts at path j iff $|f(G,S_j,i)|=1$.

If $c=0$, the proof is similar. ▮

Since we have molecular algorithm to compute any function in $\#\mathrm{Pinit}[g(n)]$ (Corollary 15), we have molecular algorithm for computing the number of Hamiltonian paths.

Theorem 19. *There is an $\{Init, Append, Merge, Separate, \mathrm{Anneal}_c, Test\}$ molecular algorithm for computing the number of Hamiltonian paths that runs in $n^{O(1)}$ time and $O(2^n n^2 \log^2 n)$ volume.*

6. Acknowledgments

We are grateful to Anne Condon for her many valuable comments, which greatly improved our presentation. We also grateful to unkonwn references for their many helpful comments.

References

[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. Baum. Dna sequences useful for computation. Technical report, NEC, 1996.

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

[6] D. Beaver. A universal 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.

[7] 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.

[8] 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.

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

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

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

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

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

[14] 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, June 1997.

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

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

[17] J. Reif. Parallel molecular computation. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 213–223, 1995.

[18] D. Roos and K. Wagner. On the power of dna-computing. *Information and Computation*, 131:95–109, 1996.

[19] 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.

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

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

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