Length Bounded Molecular Computing*

Bin Fu and Richard Beigel
Lehigh University

Abstract

Length of DNA strands is an important resource in DNA computing. We show how to decrease strand lengths in known molecular algorithms for some NP-complete problems, such as like 3-SAT and Independent Set, without substantially increasing their running time or volume.

1. Introduction

Since Adleman's pioneering experiment [1], many researchers have explored efficient molecular algorithms for NP-complete problems. The *running time* for a molecular algorithm is to the number of operations on test tubes. The *volume* is the maximum number of strings in all test tubes at any time, counting multiplicities. The *strand-length* complexity of a molecular algorithm is the length of the longest DNA strand used in the computation.

Although time and volume complexity have been well studied [13, 6, 2, 14, 5, 9, 10, 8], strand length has received less attention. Yet Roweis et al [16], state that 2500-base sequences decay at a rate of 10% per hour, and Sambrook [17] states that DNA strands longer than 10000 bases have a serious decay problem. In their survey, Ogihara et al [15] list strand-length complexity as an important resource to optimize. In this paper we show how to convert molecular algorithms for some NP-complete problems into length-efficient molecular algorithms, without substantially increasing their time or volume complexity.

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 $x = 5'\text{AGGCT}3'$, then $\overline{x} = 3'\text{TCCGA}5'$ and $\updownarrow x$ is the double strand $\substack{5'\text{AGGCT}3' \\ 3'\text{TCCGA}5'}$

2. Encoding and Operations

Let $x_1 \cdots x_n$ be an n-bit binary string. For each bit position $i \leq n$ 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 [3].

*Supported in part by the National Science Foundation under grants CCR-9700417 and CCR-9796317.
Address: Dept. of Electrical Engineering and Computer Science, 19 Memorial Dr W Ste 2, Bethlehem PA 18015-3084. Email: beigel, bif3@eecs.lehigh.edu.

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

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

Such DNA sequences are called *valid encodings.*

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 multiset of all strings in T_1 whose ith character is not c;
$T_1 := \emptyset$.

Append(T, c)

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

Merge(T_1, T_2, T_3)

$T_3 := T_1 \cup T_2$;
$T_1 := \emptyset$;
$T_2 := \emptyset$.

The implementation of those operations are described in [7].

Cut(T) $\quad T = \{x : 0x \in T \text{ or } 1x \in T\}$.

Restriction enzymes can cut double DNA strands at specific sites [12, 4, 11].

Others have proposed a variant of operation separate, which we will call Sep. It checks whether a string contains the character c anywhere. If we represent the ith symbol z_i of a string z by the symbol $\langle i, z_i \rangle$ instead, then the standard Sep operation can simulate our Separate operation with no additional overhead. The use of polynomial-size alphabets is standard practice in molecular computing. We prefer the Separate operation for convenience in programming.

We write $|x|$ to denote the *size* of a problem instance x , and we usually identify n with $|x|$.

We consider a model of $l(n)$-length, $t(n)$-time, $v(n)$-volume molecular computation. On input x, one test tube T_0 is initialized to hold encodings of the numbers $1, \ldots, v(|x|)$. A sequence of molecular operations $o_1, \ldots, o_{t(|x|)} = f(x)$ is then performed, where f is a conventional polynomial-time computable function (that is, the program is uniform in a weak but appropriate sense). The computation accepts if T_0 is nonempty after the last operation is performed.

Definition 1. MOL$^*(l(n), t(n), v(n))$ is the class of languages accepted by such a computation where the running time is $O(t(n))$ and the length of DNA strands $\leq l(n)$.

3. Simulating Circuits

Definition 2. Let π be a circuit.

- The *depth* of a gate g in π is the length of longest path from an input gate to g. The *depth* of π is the depth of π's output gate, g_{output}.

- Let $D_k(\pi)$ be the set consisting of all gates in π having depth k.

- Let $G_k(\pi)$ be the set consisting of all gates g in π such that g has depth k or less and (1) g is an input to some gate having depth greater than k or (2) g is g_{output}.
- The width of π, denoted by $w(\pi)$, is $\max_k(|G_k(\pi)|)$.

Clearly, the values of gates in $G_k(\pi)$ determine the values of all gates having depth greater than k.

Lemma 3. *Let π be a circuit with m gates. Tube T_0 contains DNA strands encoding strings in $\{0,1\}^n$. A molecular algorithm using only the operations Separate, Append, Merge, and Cut running in time $O(m)$, using DNA strands of length $O(w(\pi))$, 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.

The algorithm is divided into depth of π stages. Stage i has three phases. In phase 1, it, using the method from [7], evaluates all gates in $D_i(\pi)$ and append those values at DNA strands. In phase 2, it checks if the length of strings encoded by DNA strands is longer than $2w(\pi)$. If true, most of the gate values stored in the DNA strands will not be used in evaluating the deeper gates. We keep only the values from gates in $G_i(\pi)$, and discard the rest.

Two partial functions e_t, e_t^{-1} from N to N are constructed at stage t. We consider each DNA strand as a 0,1 string. Bit $e_t(i)$, if defined, is used to store the value of gate g_i. Gate $e_t^{-1}(j)$, if defined, is the gate that stores its value at bit j of DNA strands.

Stage 0: Let $e_0(i) = i = e_0^{-1}(i)$ and $l = n$, where l is the length of strings encoded by DNA strands in T_0.

Assume at the end of stage t, each DNA strand in T_0 has stored all values from the gates in $G_k(\pi)$.

Stage $t + 1$:

Phase 1:

$e_{t+1} := e_t$
$e_{t+1}^{-1} := e_t^{-1}$
For each gate g_i in $D_{t+1}(\pi)$ do
    Separate$(T_0, 0, e_t(j(i)), T_1, T_2)$
    Separate$(T_1, 0, e_t(k(i)), T_0, T_3)$
    Append$(T_0, f_i(0,0))$
    Append$(T_3, f_i(0,1))$
    Merge(T_0, T_3, T_1)
    Separate$(T_2, 0, e_t(k(i)), T_0, T_3)$
    Append$(T_0, f_i(1,0))$
    Append$(T_3, f_i(1,1))$
    Merge(T_0, T_3, T_2)
    Merge(T_1, T_2, T_0)
    $l := l + 1$
    $e_{t+1}(i) := l$
    $e_{t+1}^{-1}(l) := i$

Phase 2:

if $l \geq 2w(\pi)$ then
  $j := 1$
  for $i := 1$ to l do
    Separate$(T_0, 0, 1, T_1, T_2)$
    Cut(T_1)
    Cut(T_2)
    $v := e_{t+1}^{-1}(i)$
    if $v \in G_{t+1}(\pi)$ then
      Append$(T_1, 0)$
      Append$(T_2, 1)$
      $f_{t+1}(v) := j$
      $f_{t+1}^{-1}(j) := v$
      $j := j + 1$
    Merge(T_1, T_2, T_0)
  $e_{t+1} := f_{t+1}$
  $e_{t+1}^{-1} := f_{t+1}^{-1}$
  $l := j$

∎

The purpose of checking whether $l \geq 2w(\pi)$ is to control the total number steps to be $O(m)$. At most $w(\pi)$ bits are appended during Phase 1. After the Cut operations of Phase 2 are executed, strand length is no more than $w(\pi)$. Strand length is bounded by $3w(\pi)$ throughout. For a stage with $l \geq 2w(\pi)$ at the begining of Phase 2, more than $w(\pi)$ bits have been appended from the previous stage with $l \geq 2w(\pi)$ at Phase 2. Therefore, the total time in Phase 2 of all statges is propositional to the total time in Phase 1 of all stages. For each gate, it uses constant steps in Phase 1. The total time spent in Phase 1 is $O(m)$. So is the time for Phase 2. If $l \geq 2w(\pi)$, the length of strings encoded by DNA strands in T_0 is j at the end of Phase 2. Therefore, we assign j to l.

Phase 3: Let t_0 is the depth of circuit π and g_{i_0} be the output gate of π. We get the output T_1, T_2 from the following separation operation. The algorithm runs in time $O(m)$.

Separate$(T_0, 0, e_{t_0}(i_0), T_1, T_2)$;

4. The Sticker Model

Roweis et al [16] proposed a sticker model that behaves like a random memory. There are two kinds of DNA strands. Each memory model is divided into k regions, each consisting of m bases. A sticker is the (Watson–Crick) complement of a memory region. A memory region bound to a sticker encodes a 1. A memory region not bound to a sticker encodes a 0. They described a biological method to implement operations like Merge, Separate, Set (which turns on a specified bit in every strand in a tube) and Clear (which removes all stickers from a specific bit). We think each region of DNA strand is a bit, which is either 1 or 0 depending if a sticker is on.

The operations Separate(T, c, i, T_1, T_2) and Merge(T_1, T_2, T_3) are defined as in the preceding section.

Set(T, i) sets the ith bit of all strands in T equal to 1.

Clear(T, i) sets the ith bit of all strands in T equal to 0.

Lemma 4. *Let π be a circuit with m gates and e_0 be a partial function from N to N. Tube T_0 contains DNA strands encoding strings in $\{0,1\}^n$ such that for each DNA strand x, it encodes a string with the ith bit as 1 iff region $e_0(i)$ is set. Each strand in T_0 has $r \geq 3w(\pi)$ regions. A sticker Model molecular algorithm running in time $O(m)$, using DNA strands of length $O(w(\pi))$, 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 use the similar idea to preceding section in simulating circuit with many inputs. Two functions e_t and e_t^{-1} are defined as before. In other words, region $e_t(i)$ is set if gate g_i has value 1 at the end of stage t.

We will keep two sets of regions Unused and Used to store the region numbers that are occupied or non-occupied.

Stage 0: Let Used = range(e_0) and Unused = $\{1, 2, \ldots, r\}$ − Used.

Assume at the end of stage t, each DNA strand in T_0 has stored all values for the gates in $G_t(\pi)$.

Stage $t+1$:

Phase 1: In this phase we evaluate all gates in $D_{t+1}(\pi)$ and store them at unused bits in the DNA strands.

$e_{t+1} = e_t$
$e_{t+1}^{-1} = e_t^{-1}$
For each gate g_i in $D_{t+1}(\pi)$,
    choose an arbitrary $u \in$ Unused
    Unused := Unused − $\{u\}$
    Used := Used $\cup \{u\}$
    Clear(T_0, u)
    Separate($T_0, 0, e_t(j(i)), T_1, T_2$)
    Separate($T_1, 0, e_t(k(i)), T_0, T_3$)
    if $f_i(0,0)$ then Set(T_0, u)
    if $f_i(0,1)$ then Set(T_3, u)
    Merge(T_0, T_3, T_1)
    Separate($T_2, 0, e_t(k(i)), T_0, T_3$)
    if $f_i(1,0)$ then Set(T_0, u)
    if $f_i(1,1)$ then Set(T_3, u)
    Merge(T_0, T_3, T_2)
    Merge(T_1, T_2, T_0)
    $e_{t+1}(i) := u$
    $e_{t+1}^{-1}(u) := i$

Phase 2: In this phase we clear all of those bits storing values from gates not in $G_{t+1}(\pi)$. Because $|G_{t+1}(\pi)| \leq w(\pi)$, we have $|\text{Used}| \leq w(\pi)$ after the following operations, if at least $2w(\pi)$ regions are used.

if $|\text{Used}| \geq 2w(\pi)$ then
    for $i \in$ Used do
        if $e_{t+1}^{-1}(i) \notin G_{t+1}(\pi)$

then

Clear(T_0, i);

Unused := Unused $\cup \{i\}$

Used := Used $- \{i\}$

Phase 1 never use more than $w(\pi)$ regions. Thus $|\text{Used}| < 3w(\pi)$ all the time. Phase 3 is the same as that of preceding section. By the same reason at preceding section, the time is $O(m)$. ∎

We consider a $l(n)$-length, $t(n)$-time, $v(n)$-volume molecular computation in the sticker model. On input x, one test tube T_0 is initialized with $v(x)$ DNA strands of $l(|x|)$ regions to hold encodings of the numbers $1, \ldots, v(|x|)$. The first $\lfloor \log v(|x|) \rfloor + 1$ regions of each DNA strand are used to encode a number from 1 to $v(|x|)$. A sequence of molecular operations $o_1, \ldots, o_{t(|x|)} = f(x)$ is then performed, where f is a conventional polynomial-time computable function and each operation o_i is a operation in sticker model. The computation accepts if T_0 is nonempty after the last operation is performed.

Definition 5. MOL*-S$(l(n), t(n), v(n))$ is the class of languages accepted by such a computation where the running time is $O(t(n))$ and the length of DNA strands $\leq l(n)$.

5. Implementing Recursion

We slightly revise the description for the implementing recursion with bounded nondeterminism in [5].

Definition 6. A partial order $\prec$ is $(d(n), b(n))$-well-founded if

- $y_k \prec \cdots \prec y_1 \Rightarrow k \leq d(|y_1|)$; and
- $y_k \prec \cdots \prec y_1 \Rightarrow |y_k| \leq b(|y_1|)$

For technical simplicity we will consider only languages L containing the empty string, Λ.

Definition 7.

- A $(d(n), b(n), s(n), w(n), u(n))$-dsr for a language L consists of a P-uniform $s(n)$-size and $w(n)$-width computable function $h(x) = \{x_1, \ldots, x_m\}$ and a $(d(n), b(n))$-well-founded partial order $\prec$ on problem instances such that
  - Λ is the only minimal element under $\prec$
  - $(\forall x \neq \Lambda)[x \in L \iff h(x) \cap L \neq \emptyset]$
  - $(\forall x)[x_i \in h(x) \Rightarrow x_i \prec x]$
  - $|h(y)| \leq u(|x|)$ for every node y in H_x, where H_x is the least set such that $x \in H_x$ and $(y \in H_x \Rightarrow h(y) \subseteq H_x)$.
- A dsr is a $(d(n), b(n), s(n), w(n), u(n))$-dsr for some polynomials $d(n), b(n), s(n), w(n), u(n)$.

Definition 8. Let $\langle h, \prec \rangle$ be a $(d(n), b(n), s(n), w(n), u(n))$-dsr and let x be a problem instance.

- $T_{h,\prec}(x)$ is the unordered tree with root x such that, for each y, the set of children of y is $h(y)$.
- $|T_{h,\prec}(x)|$ is the number of leaves in $T_{h,\prec}(x)$.

Definition 9. A language L is in $\mathrm{REC}^*(d(n), b(n), s(n), w(n), u(n), T(x))$ if there is a $(d(n), b(n), s(n), w(n), u(n))$-dsr $\langle h, \prec\rangle$ for L such that for all x

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

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

Definition 10.

- For $n > 0$, define $\mathrm{pad}_n(x) : \{0,1\}^{\leq n} \rightarrow \{0,1\}^{2n}$ by
$$\mathrm{pad}_n(x_1 x_2 \cdots x_k) = 0x_1 0x_2 \cdots 0x_k 1^{2n-2k}.$$
- 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.
- A function f is P-uniform $s(n)$-size computable if there is a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ such that each C_n is of size at most $s(n)$ and
$$|x| \leq n \Rightarrow C_n(\mathrm{pad}_n(x)) = f(x).$$
- $L \in \mathrm{NPinit}(w(n), s(n), V(x))$ if there is a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ such that each C_n is of size at most $s(n)$ and width at most $w(n)$, and
$$x \in L \iff (\exists i)[1 \leq i \leq V(x) \text{ and } C_{|x|}(x, i) = 1].$$

Theorem 11. *If $T(x)$ is P-uniform $s'(n)$-size and $w'(n)$-width computable, then*
$$\mathrm{REC}^*(d(n), b(n), s(n), w(n), u(n), T(x)) \subseteq \mathrm{NPinit}(w^*(n), t'(n), T(x)),$$
where $t'(n) = O(d(n)(s(b(n)) + u(n)s'(b(n))))$ and $w^*(n) = O(w(n) + u(n)w'(b(n)))$.

Proof: The proof is adapted to that of a similar theorem in [5]. Let $L \in \mathrm{REC}^*(d(n), b(n), s(n), w(n), u(n), T(x))$ via $\langle h, \prec\rangle$. We will define a function $\mathrm{path}(i, x)$ taking values in $\{0, 1, \Lambda\}$ such that $\mathrm{path}(1, x) \cdots \mathrm{path}(T(x), x)$ is equal to the sequence of values at the leaves of $T_{h,\prec}(x)$ in canonical order. The function $\mathrm{path}(i, x)$ will be computed via tail recursion.

function $\mathrm{path}(i, x)$
if $x = \Lambda$ then return true
else if $h(x) = \emptyset$ then return false
else
    $\{x_1, \ldots, x_m\} := h(x)$

for $j := 1$ to m do
  if $i \leq T(x_j)$ then return path(i, x_j)
  else $i := i - T(x_j)$
return Λ

Since h is P-uniform $s(n)$-size and $w(n)$ width computable, there is a P-uniform family of circuits $\{C_{h,n}\}_{n=1}^{\infty}$ such that $C_{h,n}$ is of size at most $s(n)$, width at most $w(n)$, and $h(x) = C_{h,n}(\mathrm{pad}_n(x))$ for all x of length at most n. We have a circuit $C_1 = C_{h,b(n)}$ of size at most $s(b(n))$ such that $C_1(\mathrm{pad}_{b(n)}(y)) = h(y)$ for all y with length at most $b(n)$.

Since T is P-uniform $s'(n)$-size and $w'(n)$-width computable, there is a P-uniform family of circuits $\{C_{T,n}\}_{n=1}^{\infty}$ such that $C_{T,n}$ is of size $s'(n)$ and width $w'(n)$, and $T(x) = C_{T,n}(\mathrm{pad}_n(x))$ for all x of length at most n. So, we have a circuit $C_2 = C_{T,b(n)}$ of size at most $s'(b(n))$ such that $C_2(\mathrm{pad}_{b(n)}(y)) = T(y)$ for all y of length at most $b(n)$.

We are going to construct a circuit to compute path(x, k). $T_{h,\prec}(x)$ is a tree of depth no more than $d(|x|)$. The circuit, with input $\langle x, k\rangle$, is divided into $d(|x|)$ layers $L_0, L_1, \ldots, L_{d(|x|)}$. Layer L_0 outputs $\langle x_0, k_0\rangle = \langle x, k\rangle$. Suppose we have constructed layer L_i whose output is $\langle x_i, k_i\rangle$. Layer L_{i+1} will consist of three subcircuits, which we call $L_{i+1,1}, L_{i+1,2}$ and $L_{i+1,3}$.

- $L_{i+1,1}$ gets $\langle x_i, k_i\rangle$ from L_i, and computes $h(x_i) = \{y_1, \ldots, y_m\}$.
- $L_{i+1,2}$ gets $\{y_1, \ldots, y_m\}$ from $L_{i+1,1}$, and computes $T(y_1), \ldots, T(y_m)$.
- $L_{i+1,3}$ gets k_i, $\{y_1, \ldots, y_m\}$, and $T(y_1), \ldots, T(y_m)$ from L_i, $L_{i+1,1}$, and $L_{i+1,2}$ respectively. It computes $\langle x_{i+1}, k_{i+1}\rangle = \langle y_t, k_i - (T(y_1) + \cdots + T(y_{t-1}))\rangle$ such that $T(y_1) + \cdots + T(y_{t-1}) < k_i \leq T(y_1) + \cdots + T(y_t)$.

Clearly, $L_{i+1,1}$ has size $O(s(b(|x|)))$ and width $w(b(|x|))$, $L_{i+1,2}$ has size $O(u(|x|)s'(b(|x|)))$ and width $O(u(|x|)w'(b(|x|)))$, and $L_{i+1,3}$ has size $O(u(|x|)s'(b(|x|)))$ and width $O(u(|x|)w'(b(|x|)))$. Therefore, the size of L_i is $O(s(b(|x|)) + u(|x|)s'(b(|x|)))$. Hence, the function path can be computed by a P-uniform family of circuits with size $O(d(|x|)(s(b(|x|)) + u(|x|)s'(b(|x|))))$. and width $O(w(b(|x|)) + u(|x|)w'(b(|x|)))$. ∎

The following is a slightly revised version of a theorem in [5], which shows how to implement bounded nondeterminism with molecular computation.

Theorem 12. NPinit$(w(n), s(n), v(n)) \subseteq$ MOL$^*(w(n), s(n), v(n))$.

Proof: Let $L \in$ NPinit$(w(n), s(n), v(n))$ via a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ of size $s(n)$ and width $w(n)$ such that each C_n is of size at most $s(n)$, width at most $w(n)$, and $x \in L$ iff $C_{|x|}(x, i) = 1$ for some i with $1 \leq i \leq v(|x|)$. For each x, fix circuit $C'_x(i) = C_{|x|}(x, i)$. Applying Lemma 3, we get a molecular algorithm that uses DNA strands $O(w(n)$ and performs $O(s(|x|))$ operations determine whether there exists i such that $1 \leq i \leq v(|x|)$ and $C'_x(i) = 1$. Since $\{C_n\}_{n=1}^{\infty}$ is P-uniform, the molecular operations sequences can be also computed in polynomial time. ∎

By the same method as Theorem 12, we have a similar theorem for sticker model

Theorem 13. NPinit$(w(n), s(n), v(n)) \subseteq$ MOL*-S$(w(n), s(n), v(n))$.

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 $\max(\text{MIS}(G - v), \{v\} \cup \text{MIS}(G - v - N(v)))$

Figure 1: Tarjan's algorithm to find a maximum independent set. Herein, $d(v)$ denotes the degree of v, $N(v)$ denotes the neighbor set of v, and $\max(A, B)$ denotes the larger of the two sets A and B, with ties resolved arbitrarily.

function $h(G, k)$
pick any vertex v in G
if $d(v) \leq 1$ then
    return $\{(G - v - N(v), k - 1)\}$
else
    return $\{(G - v, k),\ (G - v - N(v), k - 1)\}$

Figure 2: A dsr corresponding to Tarjan's algorithm. Herein, $d(v)$ denotes the degree of v and $N(v)$ denotes the neighbor set of v.

5.1. Implementing Algorithms for Independent Set and 3SAT

An independent set in a graph is a set of nodes, no two of which are adjacent. A famous NP-complete problem, the *independent set problem* is the following: Given a graph G and a positive integer k, is there an independent set in G consisting of at least k vertices?

Tarjan [18] gave a simple algorithm to find the largest independent set in a graph. We present Tarjan's algorithm in Figure 1. A simple modification yields a dsr for the independent set problem. We present the corresponding dsr in Figure 2.

In our complexity bounds, n denotes the number of vertices in G, e denotes the number of edges, and $|G|$ denotes $\max(e, n)$. Define $G_1 \prec G_2$ if the number of vertices in G_1 is less than the number of vertices in G_2. We claim that $\langle h, \prec \rangle$ is a $(O(n), O(|G|), O(|G|), O(|G|), 2)$-dsr with at most $t(n)$ leaves, where $t(n)$ satisfies $t(n) = t(n-1) + t(n-3)$. Let $T(G) = t(m(G))$ where $m(G)$ is the number of vertices in G. Clearly, h, $\prec$, and $T()$ satisfy the conditions of Definition 8. The function h is P-uniform $O(|G|)$-size and $O(|G|)$-width computable. The partial order $\prec$ is $(O(n), O(|G|))$-well-founded. The recurrence can be solved explicitly to obtain $t(n) < 1.47^n$.

We implement Tarjan's algorithms for independent set problem and have the following theorem. We decrease the length complexity to $O(n)$ and keep the same time and volume complexity. This improves previous algorithms for this problem [2, 5].

Theorem 14. *There is a* {Separate, Merge, Append, Cut, Init}*-molecular algorithm for Independent set problem with* $O(|G|)$*-length, O(n—G—)-time and* 1.47^n*-volume.*

Implementing Monien and Speckenmeyer's algorithm for 3SAT problem, we have the following theorem. The length used in their paper is $O(mn)$, where n is the number of variables and m is the number of clauses. we decrease the length complexity to $O(m)$ while keep the same order complexity for time and volume, This improve the results in [14, 5].

Theorem 15. *There is a* $\{\text{Separate}, \text{Merge}, \text{Append}, \text{Cut}, \text{Init}\}$*-molecular algorithm for 3SAT problem with* $O(m)$*-length,* $O(mn)$*-time and* 1.62^n*-volume.*

Applying Theorem 13, we also have similar results for implementing the two algorithms in sticker model.

6. Acknowledgment

We are grateful to Mitsu Ogihara for sending his survey [15], which is very helpful for this research, and unknown reference for many helpful comments for an earlier version of this paper.

References

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

[2] 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.

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

[4] D. Beaver. Computing with dna. *Journal of Computational Biology*, 2(1), 1995.

[5] 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.

[6] 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.

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

[8] B. Fu. *Volume Bounded Molecular Computation.* PhD thesis, Yale University, Dept. of Computer Science, New Haven, CT, 1997.

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

[10] B. Fu, R. Beigel, and F. Zhou. An $O(n^2(\log^2 n)2^n)$ volume molecular algorithm for Hamiltonian path. Manuscript, 1997.

[11] T. Head. Formal language theory and dna: An analysis of the generative capacity of recombinant behaviors. *Bull. Math. Biol.*, 49:737–759, 1987.

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

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

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

[15] M. Ogihara, A. Ray, and K. Smith. Biomolecular computing—a shape of computation to come. *SIGACT News*, 28(3):2–11, 1997.

[16] 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.

[17] J. Sambrook, E. F. Fritsch, and T. Maniatis. *Molecular Cloning: a Laboratory Manual*, volume 1. Cold Spring Harbor Press, New York, 2nd edition, 1989.

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