Yale University
Department of Computer Science

Molecular Computing, Bounded Nondeterminism, and Efficient Recursion

Richard Beigel Bin Fu
Yale University and University of Maryland

YALEU/DCS/TR-1116
November 1996

Molecular Computing, Bounded Nondeterminism, and Efficient Recursion *

Richard Beigel[†] Bin Fu[‡]

Yale University and University of Maryland

Abstract

The maximum number of strands used is an important measure of a molecular algorithm's complexity. This measure is also called the *space* used by the algorithm. 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 space $2^{b(n)}$. 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-colorability.

1. A model of molecular computing

Molecular computation was first studied in [1, 17]. The models we define were inspired as well by the work of [3, 23]. 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 multi-set of molecular sequences. We describe the allowable operations below. Where set notation is applied to multi-sets, multiplicities are respected. In the definitions T_1, T_2, and T_3 denote distinct test tubes, c denotes a character, and i denotes a positive integer.

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;

$T_1 := \emptyset$.

Pour(T_1, T_2)

$T_2 := T_1$;

$T_1 := \emptyset$.

---

*Both authors are visiting U. Maryland 1996–97. Address: Dept. of Computer Science, University of Maryland at College Park, College Park, MD 20742-3251. Research supported in part by the National Science Foundation under grants CCR-8958528 and CCR-9415410.

[†]Also supported in part by NASA under grant NAG 52895. Email: `beigel@cs.umd.edu`

[‡]Email: `binfu@umiacs.umd.edu`

Append(T, c)
$T := \{xc : x \in T\}.$

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

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.

The running time for a molecular algorithm is proportional to the number of operations on test tubes. An important complexity measure is the *solution space size* (also called simply *space*), i.e., 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} might be possible. Recent papers [3, 19] attempt to optimize solution space size for particular combinatorial problems.

Problem instances are associated with a parameter n called their size. In complexity theory, n is the length of a suitable encoding of the instance. However, in analysis of algorithms, n is usually a more natural representation-independent parameter, such as the number of vertices in a graph or number of variables in a formula. Although the n's of complexity theory and the n's of analysis of algorithms are usually polynomially related, it can make a phenomenal difference when n appears in the exponent. For that reason we take n to be a problem-dependent but representation-independent notion of size through this paper. We write $|x|$ to denote the *size* of a problem instance x rather than its length, and we usually identify n with $|x|$.

We consider a highly restricted model of $t(n)$-time, $s(n)$-space molecular computation, which we think has a good chance of eventually being practical. On input x, one test tube T_0 is initialized to hold encodings of the numbers $1, \ldots, s(|x|)$. 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. MOL$(s(n))$ is the class of languages accepted by such a computation where the running time $t(n)$ is polynomial bounded.

2. Results for Particular Problems

Problem \ Results	Previously			In This Paper	
	Space	Limited Model	Reference	Space	Limited Model
Hamiltonian Path	$n!$	$\surd$	[1]		
SAT	2^n	$\surd$	[17]		
QBF	2^n	$\times\times$	[23]		
3-SAT	1.62^n	$\times$	[19]	1.50^n	$\surd$
3-Colorability	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	$\surd$

3. Bounded Nondeterminism

NP computation with a limited amount of nondeterminism was introduced in [14, 15, 16] and studied further in [10, 11, 20, 9, 12, 25, 13, 7]. The class NPbits($b(n)$) consists of languages recognized by an NP machine that make at most $b(n)$ binary nondeterministic choices on each computation path on inputs of size n. (Actually, prior treatments allowed $O(b(n))$ binary choices, but the constant factor turns out to be very important in connection with molecular computation.) We define a refinement of this class: NPinit($s(n)$) consists of languages recognized by NP machines that nondeterministically choose a number between 1 and $s(n)$ on inputs of size n and then behave deterministically. Clearly, NPbits($b(n)$) = NPinit($2^{b(n)}$).

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

In this section we show how simulate to bounded nondeterministic computation via bounded-space molecular computation. Results of this type appear in [4, 23, 24, 29], 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. [8] obtained a result similar to ours.

Lemma 1. *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:

for $i := n+1$ to m do
    Separate($T_0, 0, j(i), T_1, T_2$)
    Separate($T_1, 0, k(i), T_0, T_3$)
    Append($T_0, f_i(0,0)$)
    Append($T_3, f_i(0,1)$)
    Merge(T_0, T_3, T_1)
    Separate($T_2, 0, k(i), T_0, T_3$)
    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)
Separate($T_0, 0, m, T_1, T_2$)

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

Theorem 2. 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 [21]). Apply Lemma 1 to π to see that L is in MOL($s(n)$). ∎

5. 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 ≤ (n−1 choose k) then
        Enum(i, x0, n − 1, k)
    else
        Enum(i − (n−1 choose 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.

5.1. 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 [27] and d-self-reductions were defined in [28].

Definition 3. 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 4. 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 5. 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 6. Let T be a polynomial-time computable function. A language L is in $\mathrm{REC}(T(x))$ 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 7. $\mathrm{NPinit}'(S(x))$ 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 $\mathrm{NPinit}'(S(x)) \subseteq \mathrm{NPinit}(s(n))$.

Theorem 8. $\mathrm{REC}(T(x)) \subseteq \mathrm{NPinit}'(T(x))$.

Proof: Let $L \in \mathrm{REC}(T(x))$ via $\langle h, \prec \rangle$. We will define a deterministic polynomial-time computable 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 proof is completed by having the ith path of an $\mathrm{NPinit}'(T(x))$ machine compute $\mathrm{path}(i,x)$; clearly that machine accepts L. The function $\mathrm{path}(i,x)$ will be computed via tail recursion.

```
function path(i, x)
if x = Λ 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 9. We say that a partial order $\prec$ on problem instances is *parameterizable* if there are 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 10. 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 11. 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 12. *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 9, $|\{i : i \prec' m(x)\}| \leq p(|x|)$. 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. ∎

5.2. 3-SAT

In this section, we apply our results to the classic 3-SAT algorithm of Monien and Speckenmeyer [18] and a recent unverified 3-SAT algorithm of Schiermeyer [26]. 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.

5.2.1. Monien and Speckenmeyer's Algorithm

The size of a satisfiability instance is the number of variables. Consider the 3-SAT algorithm of Monien and Speckenmeyer. 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|_v̄)
else if f contains a clause C consisting of a single literal ℓ then
    return 3SAT(f|_ℓ)
else if f contains a clause C consisting of two literals ℓ1, ℓ2 then
    return 3SAT(f|_ℓ1) ∨ 3SAT(f|_ℓ̄1|_ℓ2)
else
    let v be the first variable to appear in f
    return 3SAT(f|_v) ∨ 3SAT(f|_v̄)
```

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 12 and 3-SAT is in $\mathrm{REC}(T_0(F))$ for some $T_0 \leq T_{3\mathrm{S}}$. By Theorem 8 and Theorem 2, $3\mathrm{SAT} \in \mathrm{NPinit}(t_{3\mathrm{S}}(n))$, so 3-SAT is in $\mathrm{MOL}(1.62^n)$. The same space bound for 3-SAT was obtained previously by Ogihara [19], but in a model that allows more powerful operations like Polymerization, which can implement the Amplify operation.

5.2.2. Schiermeyer's Algorithm

Schiermeyer [26] reports a 1.497^n time algorithm for 3-SAT problem. His algorithm is a d-self-reduction for the 3-SAT problem. We will prove that 3SAT $\in$ REC($T(F)$), where the function $T(F) \leq 1.497^n$ and will be defined below. We follow [26] 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 6. Hence, 3SAT $\in$ REC($T(F)$) $\subseteq$ MOL(1.497^n).

5.3. 3-Coloring and $(3, 2)$-System

Beigel and Eppstein [6] give algorithms for $(3, 2)$-system 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. $(3, 2)$-system generalizes 3-coloring, 3-SAT and 3-edge-coloring.

5.3.1. $(3, 2)$-System Algorithm

The size of a $(3, 2)$ system is the number of variables in it. Beigel and Eppstein's [6] $(3, 2)$-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, $\text{mh}(F) = \{n - (4 + i), n - 1\}$, where $i \geq 0$.

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

In cases 2b, 2d, 6, 8c and 9, $\text{mh}(F) = \{n'\}$, 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 12, 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 6. Thus, $(3,2)$-system is in $\mathrm{REC}(T_0(F))$. So,

$$\begin{array}{rcll}
(3,2)\text{-system} & \in & \mathrm{NPinit}'(T_0(F)) & \text{by Theorem 8} \\
& \subseteq & \mathrm{NPinit}'(T(F)) & \text{because } T_0(F) \leq T(F) \\
& = & \mathrm{NPinit}(t(n)) & \text{because } T(F) = t(|F|) \\
& = & \mathrm{NPinit}(1.38028^n) & \text{because } t(n) = 138028^n \\
& \subseteq & \mathrm{MOL}(1.38028^n) & \text{by Theorem 2.}
\end{array}$$

5.3.2. 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 $\mathrm{NPinit}(1.345^n)$ 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 $\mathrm{32SYS}(F)$ using the nondeterministic choices dictated by b

Therefore 3-coloring is in $\mathrm{MOL}(1.345^n)$.

5.4. 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?"

5.4.1. Tarjan's Algorithm

Consider the following simple algorithm due to Tarjan [30]. ($d(v)$ denotes the degree of v, and $N(v)$ denotes the neighbor set of v. $\max(S, T)$ 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 $\{v\} \cup \mathrm{MIS}(G - v - N(v))$
else
    return $\max(\mathrm{MIS}(G - v), \{v\} \cup \mathrm{MIS}(G - v - N(v)))$

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 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.

5.4.2. Robson's Algorithm

The best published purely recursive algorithm for the independent set problem is due to Robson [22] 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 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 [5] that is based on a d-self-reduction and hence is directly adaptable to molecular computing.

6. Acknowledgments

We are grateful to William Gasarch for his patience in reading this paper as well as his suggestions in improving the presentation. We are also grateful to Tirza Hirst for helpful discussions and to Ingo Schiermeyer for sharing with us a preliminary draft of [26].

References

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

[2] L. Adleman. On construction a molecular computer. In *1st DIMACS workshop on DNA Computing*, 1995.

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

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

[6] 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.

[7] 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.

[8] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. Manuscript, 1996.

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

[10] 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.

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

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

[13] L. Hemachandra and S. Jha. Defying upward and downward separation. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 185–195. Springer-Verlag, 1993. LNCS 665.

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

[15] 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.

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

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

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

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

[20] 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.

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

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

[23] D. Roos and K. Wagner. On the power of bio-computers. Tr, University of Wurzburg, Feb. 1995. ftp://haegar.informatik.uni-wuerzburg.de/pub/TRs/ro-wa95.ps.gz.

[24] P. Rothemund. A DNA and restriction enzyme implementation of Turing machines. http://www.ugcs.caltech.edu/~pwkr/oett.html.

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

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

[27] 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.

[28] A. L. Selman. Natural self-reducible sets. Tr, Northeastern University, Boston, MA 12115, 1986.

[29] W. Smith and A. Schweitzer. DNA computers in vitro and vivo. Tr, NEC, 1995.

[30] R. Tarjan. Finding a maximum clique. TR 72-123, Cornell University, 1972.

Appendix

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), $\ldots$, $b_m +$ 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 [22], we know that

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

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 $k \le b_i$ then
return true
else
return $\bigvee_{i=1}^{m} \mathrm{IND}(F_i, k - b_i)$

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 $k \le b_i$ 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 $\mathrm{IND}(\langle G, \emptyset, 0 \rangle, k) =$ true. It is easy to see that

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

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

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

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

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

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