# A Comparison of Resource-Bounded Molecular Computation Models*

Bin Fu and Richard Beigel
Lehigh University

March 24, 1998

**Abstract**

The number of molecular strands used by a molecular algorithm is an important measure of the algorithm's complexity. This measure is also called the *volume* used by the algorithm. We prove that three important polynomial-time models of molecular computation with bounded volume are equivalent to models of polynomial-time Turing machine computation with bounded nondeterminism. Without any assumption, we show that the Split operation does not increase the power of polynomial-time molecular computation. Assuming a plausible separation between Turing machine complexity classes, the Amplify operation does increase the power of polynomial-time molecular computation.

## 1. Introduction

Molecular computation was first studied in [1, 15], which identified the number of molecular strands used as an important resource. This measure is also

*Research performed at Yale University and at the University of Maryland. Supported in part by the National Science Foundation under grant CCR-8958528, CCR-9415410, and CCR-9700417. The first author was also supported in part by NASA under grants NAG 52895. Address: Dept. of Electrical Engineering and Computer Science, 19 Memorial Dr W Ste 2, Bethlehem PA 18015-3084. Email: {beigel bif3}@eecs.lehigh.edu. Web: http://www.eecs.lehigh.edu/∼{beigel bif3}.

called the *volume* (or the size of the solution space) used by the algorithm. Recent papers [3, 16, 4] have attempted to optimize solution space size for particular combinatorial problems. In this paper we begin a systematic study of volume-bounded molecular computation, and we determine the relations among several models of molecular computation and Turing machine computation.

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

We consider three polynomial-time, $s(n)$-volume bounded models, all of which permit the operations Separate, Pour, Append, and Merge (all operations will be defined formally in Section 2): in the MOL($s(n)$) model one tube $T_0$ is initialized to hold encodings of the numbers $1, \ldots, s(n)$;[1] in the MOL$'$($s(n)$) model, several tubes are initialized to contain a total of $s(n)$ copies of the empty string, and the Split operation is also permitted; in the MOL-A($s(n)$) model, $T_0$ is initialized to contain a single copy of the empty string, the Amplify operation is permitted, and the total number of strings in all tubes must be at most $s(n)$ at all times. All tubes other than $T_0$ are initialized to be empty. The molecular algorithm *accepts* if $T_0$ is nonempty after the last molecular operation is performed.

We consider two polynomial-time, $s(n)$-path bounded models of nondeterministic computation: an NPinit($s(n)$) machine nondeterministically chooses a number between 1 and $s(n)$ and then proceeds deterministically; an NPpaths($s(n)$) machine may perform arbitrary nondeterministic computation provided that the total number of computation paths is at most $s(n)$. We prove

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

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

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

Finally, we note that Winfree [21] has proved some relations between branching programs and memoryless molecular computation (where the Append operation is not allowed). His results suggest that the Amplify operation might increase the power of molecular computation. However, he considers only time complexity and not volume complexity for molecular computation. Furthermore, only brute-force algorithms have been implemented in the memoryless models he describes.

# 2. Molecular Operations

Molecular computation was first studied in [1, 15]. The models we define were inspired as well by the work of [3, 19]. A molecular sequence is a string over an alphabet $\Sigma$ (we can use any alphabet we like, encoding characters of $\Sigma$ by finite sequences of base pairs). A test tube is a multiset of molecular sequences. We describe the allowable operations below. Where set notation is applied to multisets, multiplicities are respected. In the definitions, $T_1$, $T_2$, and $T_3$ denote distinct test tubes and $c$ denotes a character.

**Sep**$(T_1, c, T_2, T_3)$

$T_2 :=$ the multiset of all strings in $T_1$ that contain the character $c$ anywhere;
$T_3 :=$ the multiset of all strings in $T_1$ that do not contain the character $c$ anywhere;
$T_1 := \emptyset$.

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

$T_2 :=$ the multi-set of all strings in $T_1$ whose $i$th char-

acter is $c$;
$T_3 :=$ the multi-set of all strings in $T_1$ whose $i$th character is not $c$;
$T_1 := \emptyset$.

**Pour**$(T_1, T_2)$

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

**Append**$(T, c)$

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

**Merge**$(T_1, T_2, T_3)$

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

**Split**$(T_1, T_2, T_3)$

$T_2$ and $T_3$ are obtained by halving the multiplicities of all strings in $T_1$, which are assumed to be even;
$T_1 := \emptyset$.

**Amplify**$(T_1, T_2, T_3)$

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

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

The 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," i.e., the maximum number of strings in a tube, counting multiplicities. Adleman [2] has speculated that molecular computation with a solution space of size $2^{70}$ might be possible. Recent papers [3, 16] attempt to optimize solution space size for particular combinatorial problems.

## 2.1. Example: satisfying a simple Boolean formula

This example, finding all satisfying assignments to the formula $(x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$, is from [15]. We will show how to solve this problem in the MOL(), MOL′(), and MOL-A() models.

MOL(4) **Algorithm**

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

MOL′(4) **Algorithm**

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

| Initialization | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|---|---|
| | $\Lambda, \Lambda, \Lambda, \Lambda$ | | | | |
| Split($T_0, T_1, T_2$) | | $\Lambda, \Lambda$ | $\Lambda, \Lambda$ | | |
| Append($T_1, 0$) | | $0, 0$ | $\Lambda, \Lambda$ | | |
| Append($T_2, 1$) | | $0, 0$ | $1, 1$ | | |
| Merge($T_1, T_2, T_0$) | $0, 0, 1, 1$ | | | | |
| Split($T_0, T_1, T_2$) | | $0, 1$ | $0, 1$ | | |
| Append($T_1, 0$) | | $00, 10$ | $0, 1$ | | |
| Append($T_2, 1$) | | $00, 10$ | $01, 11$ | | |
| Merge($T_1, T_2, T_0$) | $00, 10, 01, 11$ | | | | |

MOL-A(4) **Algorithm**

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

| Initialization | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|---|---|
| | $\Lambda$ | | | | |
| Amplify($T_0, T_1, T_2$) | | $\Lambda$ | $\Lambda$ | | |
| Append($T_1, 0$) | | $0$ | $\Lambda$ | | |
| Append($T_2, 1$) | | $0$ | $1$ | | |
| Merge($T_1, T_2, T_0$) | $0, 1$ | | | | |
| Amplify($T_0, T_1, T_2$) | | $0, 1$ | $0, 1$ | | |
| Append($T_1, 0$) | | $00, 10$ | $0, 1$ | | |
| Append($T_2, 1$) | | $00, 10$ | $01, 11$ | | |
| Merge($T_1, T_2, T_0$) | $00, 10, 01, 11$ | | | | |

# 3. Bounded Nondeterminism

NP computation with a limited amount of nondeterminism was introduced in [12, 13, 14] and studied further in [8, 9, 17, 7, 10, 20, 11, 5]. The class NPbits($b(n)$) consists of languages recognized by NP machines that make at most $b(n)$ binary nondeterministic choices on each computation path on inputs of length $n$. (Actually, prior treatments allowed $O(b(n))$ binary choices, but the constant factor is very important in connection with molecular computation, because it will appear in the exponent.) We define two related

classes: NPinit($s(n)$) consists of languages recognized by NP machines that nondeterministically choose a number between 1 and $s(n)$ on inputs of length $n$ and then behave deterministically. NPpaths($s(n)$) consists of languages recognized by NP machines that have at most $s(n)$ computation paths on each input of length $n$. Two easy observations:

- NPbits($b(n)$) = NPinit($2^{b(n)}$)
- NPinit($s(n)$) $\subseteq$ NPpaths($s(n)$)

Thus the NPinit() hierarchy refines the NPbits() hierarchy. It is unknown whether NPinit($s(n)$) = NPpaths($s(n)$).

# 4. MOL$'(s(n))$ = MOL$(s(n))$ = NPinit$(s(n))$

First, it is easy to see that MOL($s(n)$) $\subseteq$ MOL$'(s(n))$. Write $s(n) = 2^{a_1} + \cdots + 2^{a_k}$ where $a_1 > \cdots > a_k$, and initialize tube $T_i$ to contain $2^{a_i}$ copies of the empty string for $i = 1, \ldots, k$. Then use Split, Append, and Merge to generate the numbers $2^{a_1} + \cdots + 2^{a_{i-1}} + 1, \ldots, 2^{a_1} + \cdots + 2^{a_i}$ in tube $T_i$. Second, Boneh et al [6] and independently Beigel and Fu [4] have shown that NPinit($s(n)$) $\subseteq$ MOL($s(n)$). In this section we will show that MOL$'(s(n))$ $\subseteq$ NPinit($s(n)$). Thus MOL$'(s(n))$ = MOL($s(n)$) = NPinit($s(n)$).

**Theorem 1.** MOL$'(s(n))$ $\subseteq$ NPinit($s(n)$)

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

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

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

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

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

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

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

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

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

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

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

**Corollary 2.** MOL$'(s(n))$ = MOL$(s(n))$ = NPinit$(s(n))$

Note 1: Bach et al [3] define an asymmetric split operation that splits $T_1$ in a specified ratio $r : 1-r$. An ordinary split cannot simulate even an asymmetric split because an ordinary split is undefined on a tube with an odd number of elements. However, our proof of Theorem 1 works for asymmetric splits (just replace one half by the fraction $r$). Thus asymmetric splits are no better than ordinary splits as far as volume is concerned.

Note 2: In our simulation and in [4, 6] there is a 1–1 correspondence between molecular sequences and computational paths. Thus the counting versions of the classes above also coincide.

# 5. MOL-A$(s(n))$ = NPpaths$(s(n))$

**Lemma 3.** MOL-A$(s(n)) \subseteq$ NPpaths$(s(n))$

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

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

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

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

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

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

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

**Lemma 4.** NPpaths$(s(n)) \subseteq$ MOL-A$(s(n))$

**Proof:** Without loss of generality, assume that the NPpaths$(s(n))$ machine makes only binary choices. Call its input $x$. Modify the machine so that it behaves deterministically — all choices specified by an auxiliary binary input $y$. Convert that machine to an equivalent circuit by the techniques of [18]. For a given $x$, we wish to evaluate that circuit for all $s(|x|)$ possible $y$ using molecular computation. The molecular algorithm evaluates the circuit bottom up. If a gate depends on the $j$th bit of $y$ then the algorithm first separates sequences in which the value of this bit is not already stored. It amplifies those sequences, appends $\langle j, 0\rangle$ to half of them and $\langle j, 1\rangle$ to the other half, merges the two halves, and then merges in the sequences in which the bit was already stored. If the gate depends on two bits of $y$, then this procedure is repeated for the second bit. At this point the gate's value is determined, and can be computed using the techniques of [4, 6]. ∎

Thus we have

**Theorem 5.** NPpaths$(s(n)) =$ MOL-A$(s(n))$

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

# 6. Oracles separating NPinit$(s(n))$ from NPpaths$(s(n))$

We conjecture that NPinit$(s(n)) \subset$ NPpaths$(s(n))$. Let $f(n) = o(2^{2n}/n^k)$ for every $k$. To support our conjecture, we construct an oracle $A$ such that

$\mathrm{NPpaths}(2^{n+1}-1)^A \not\subseteq \mathrm{NPinit}(f(n))^A$. For each set $A$, define

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

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

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

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

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

Define

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

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

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

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

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

# 7. Open Problems

We have shown that MOL($s(n)$) = MOL-A($s(n)$) iff NPinit($s(n)$) = NPpaths($s(n)$). Obviously, if P = NP then NPinit($s(n)$) = NPpaths($s(n)$). How does NPinit($s(n)$) $\stackrel{?}{=}$ NPpaths($s(n)$) relate to other questions in complexity theory? A large class of recursive algorithms can be converted to NPinit($s(n)$) algorithms [4]. It would be interesting to look for a larger class of algorithm thats could be converted to NPpaths($s(n)$) algorithms.

Our simulation of MOL$'$($s(n)$) by NPinit($s(n)$) maps $t(n)$-time molecular algorithms to $t(n)^4$-time RAM algorithms (we have not closely examined the Turing machine implementation of these algorithms). Beigel and Fu's [4] and Boneh et al's [6] simulation of NPinit($s(n)$) by MOL($s(n)$) maps $t(n)$-time Turing machine algorithms to $t(n) \log t(n)$-time molecular algorithms, because of the overhead for converting a Turing machine to a circuit. Can the simulations be performed more efficiently, say at the cost of a constant factor in running time?

# 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] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] 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.

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

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

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

[17] 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.

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

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

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

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