

DNA Models and Algorithms for NP-complete Problems *

Eric Bach Anne Condon
Elton Glaser Celena Tanguay

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 57306 USA

Abstract

A goal of research on DNA computing is to solve problems that are beyond the capabilities of the fastest silicon-based supercomputers. Adleman and Lipton present exhaustive search algorithms for 3-Sat and 3-Coloring, which can only be run on small instances and hence are not practical.

In this paper, we show how improved algorithms can be developed for the 3-Coloring and Independent Set problems. Our algorithms use only the DNA operations proposed by Adleman and Lipton, but combine them in more powerful ways, and use polynomial preprocessing on a standard computer to tailor them to the specific instance to be solved. The main contribution of this paper is a more general model of DNA algorithms than that proposed by Lipton. We show that DNA computation for NP-complete problems can do more than just exhaustive search. Further research in this direction will help determine whether or not DNA computing is viable for NP-hard problems. A second contribution is the first analysis of errors that arise in generating the solution space for DNA computation.

*Supported in part by NSF grant numbers CCR 95-10244 and CCR-9257241 and by matching awards from Thinking Machines Corporation and Digital Systems Corporation. E-mail addresses of the authors are: bach@cs.wisc.edu, condon@cs.wisc.edu, glaserea@cs.wisc.edu, tanguay@cs.wisc.edu.

1 Introduction

Adleman described how he used standard tools of molecular biology to solve a 7-vertex instance of the Hamiltonian Path problem [1]. A major goal of subsequent research in this area is to understand how DNA computing can be used to solve NP-hard problems.

To address this goal, Lipton [8] and Adleman [2] proposed the following model of DNA computation. A molecular computation proceeds in two phases: solution space generation and computation. The solution space generation phase yields a test tube whose contents (DNA strands) encode strings over an alphabet Σ , whose size can depend on the input. Throughout, we consider a test tube to be a multiset of strings. In each step of the computation phase, an operation is performed on one or more test tubes. The **separate** operation takes one test tube and a symbol $\sigma \in \Sigma$ and returns two test tubes, one containing the set of strings containing σ and the other containing the remaining strings. The **merge** operation forms the union of two test tubes. The **test-if-empty** operation tests whether a test tube is empty. Adleman suggests that the **test-if-empty** operation be used only at the last step of the computation.

To construct the test tube of strings in the solution space generation phase, the following operations are applied to a test tube initially containing empty strings. The **append**(σ) opera-

tion appends the symbol σ to all strands in a test tube. The **split** operation partitions a test tube into two such that for each distinct string x in the test tube, if there are l copies of x , then there are $l/2$ copies in each of the two test tubes. (This can be generalized to a split into k test tubes). These operations are used in a very restricted way by Lipton to generate a solution space that represents all possible binary strings of length n and by Adleman [2] to generate a solution space that represents all 3^n possible colorings of n vertices using 3 colors.

Lipton [8] and Adleman [2] present simple molecular algorithms for 3Sat and 3-Coloring, with solution spaces of size 2^n and 3^n , respectively. In contrast, the best exact algorithms for standard computing models avoid searching through the whole solution space. Biegel and Eppstein have a $O(1.35^n)$ algorithm for 3-Coloring and Schiermeyer [10] has a $O(1.58^n)$ algorithm for 3Sat. The naive DNA algorithms can't expect to beat these algorithms on any instance size. To see this, suppose we accept Adleman's speculation that a solution space of size 2^{70} can be used in a DNA computation. Then, for the 3-coloring problem, the largest possible instance solvable by a DNA computation has 44 vertices. Moreover, if this instance has, say, 400 edges, it would take well over 10 days to perform the DNA computation, assuming optimistically that the average DNA operation takes only 20 minutes (again, from Adleman). In contrast, if we estimate the number of operations for Biegel and Eppstein's algorithm at 1.35^n , that is, we ignore the constant in the running time, then on a (slow!) computer that can perform 10^6 operations per second, a 44-vertex instance of 3-Coloring can be solved in just over half a second, and a 70-vertex instance can be solved in under 22 minutes.

In this paper, we show that if the operations proposed by Adleman and Lipton are combined in more general ways, algorithms for 3-Coloring and Independent Set can be obtained with a much smaller solution space. Our results show that

DNA computations can be used for more than simple exhaustive search. Hence, it is premature to reject DNA computing on the basis of the impracticality of exhaustive search. In order to determine if DNA computing is practical, it is important to design and analyze the best possible algorithms we can for this paradigm. Our results represent a first step in this direction. As a bonus, we also get some new and different parallel programming ideas by considering this model. They might be incorporated into future computers that do not use DNA.

The main differences between our model and that of Lipton are in the solution space generation phase. We define our new model carefully in Section 2. In our model, the solution space can be pruned and tailored to the problem instance, avoiding generation of all the combinatorial possibilities. Efficient methods for generating solution spaces other than the set of combinatorial possibilities are interesting in their own right and methods for doing this are already used in combinatorial chemistry [5, 6].

In Section 3, we present our new algorithms. All of our algorithms do a polynomial number of molecular operations. Each algorithm is named by the size of its solution space. For each, we list the differences between our model and that of Lipton that allow us to obtain improved bounds on the solution space size.

A $n^{1.89^n}$ 3-Coloring Algorithm: To obtain this algorithm, we allow test tube contents to be split into weighted subsets and allow splits of already split test tubes in the solution space generation phase. In this way, we can generate the set of strings representing all possible subsets of at most $n/3$ elements of a set of size n . Our algorithm then simply checks if there is an independent subset of vertices of size at most $n/3$, such that the remaining graph is bipartite. The **append** operation is used in the computation phase.

A 1.67^n Independent Set Algorithm: This algorithm uses a polynomial-time preprocessing phase (to be done on a standard computer) that tailors the solution space to the instance to be

solved. The computation phase is very simple; the append operation is not even needed.

A 2^n 3-Coloring Algorithm: Although this algorithm has an asymptotically larger solution space than the $n1.89^n$ algorithm, the computation phase is much simpler. A new feature of this algorithm is the use of the **separate** operation in a restricted way in the solution space generation phase, in addition to the **append** and **split** operations.

A 1.51^n Independent Set Algorithm: In this algorithm, computation on a standard computer is interleaved with DNA computation throughout. As in the $n1.89^n$ 3-Coloring algorithm, weighted splits and repeated splits of test tubes are needed.

The second contribution of the paper is analysis of errors due to an imperfect **split** operation. Lipton's solution space generation model implicitly assumes that in the **split** operation, every subset of identical strands in the test tube is split perfectly in halves. For example, if a test tube contains 20 strings, say 10 copies each of strings s_1 and s_2 , then 5 copies of each string appear in each of the two test tubes resulting from the split. In reality however, a **split** of a test tube is implemented by pouring equal amounts of the contents of a test tube into two test tubes [2]. Even if we assume that the total number of strings in the two test tubes is equal, we can't expect to get a perfect split of each subset of identical strings. Imperfect splits can cause some strings not to be present in the final solution space.

To model this, we replace the **split** operation with a **probabilistic-split** operation, in which each possible partition of the test tube into halves is equally likely. This probabilistic model may seem to be an oversimplification. It corresponds, however, to the use of the most probable distribution in statistical mechanics, which usually leads to correct answers. (For example, the Boltzmann distribution is usually derived heuristically in this way.) To ensure that all strings in the solution space are present with high probability, one can place in the initial test tube a number of strings

that is many times greater than the size of the desired solution space. To make this precise, for a given error parameter ϵ , we define the *redundancy* of the solution space generation phase to be an integer such that if the number of strings in the initial test tube is equal to the solution space size times the redundancy, then with probability $1 - \epsilon$, the final test tube contains all strings in the desired solution set. This method avoids the need for amplification of strands, which Adleman suggests should be avoided whenever possible.

In Section 4, we analyze the redundancy that is necessary for the solution space generation phases used in our algorithms. To do this, we introduce a new model (graphs with urns) to model the DNA solution space generation process.

We first consider Lipton's model for generating a solution space of size k^n , and show that a redundancy of $\Theta(n)$ guarantees success with all but exponentially small probability. The same is true of the solution sets used in our 1.67^n Independent Set and 2^n 3-Coloring algorithms. The solution space of the remaining algorithms is the set of all binary strings of length n with exactly (or at most) k 1's. We can show that a redundancy of n^7 is sufficient for these solution spaces. We believe that this is an overestimate, however.

2 A Model of DNA Computation

2.1 Solution Space Generation Phase

We model the solution space generation algorithm, for a given instance of a problem, using a finite dag called a *generator*. Associated with each edge of the generator is a test tube. The generator and associated test tubes are defined as follows. The graph has a root vertex with out-degree one and a sink vertex with indegree one. All vertices are reachable from the root vertex and the sink vertex can be reached from all vertices. The vertices of the graph are partitioned into levels, with all edges leaving vertices at level

i going to vertices at level $i + 1$. Each edge has a positive rational weight (probability), such that the sum of the weights out of a vertex is 1.

Each vertex is labeled by one of the operations **merge**, **separate**, **append(symbol)**, **split** or **no-op**. The vertices and multisets associated with the edges are constrained as follows. First, the set associated with the edge leaving the root is a multiset of empty strings. The size of this set is defined to be the size of the solution space generated by the generator.

A **split** vertex has indegree 1 and arbitrary outdegree. Let the multiset associated with the incoming edge be M' . The multiset M associated with an outgoing edge of weight p is defined as follows. For each string s in M' with multiplicity m , M contains pm copies of s . If pm is not an integer, the graph is not a valid generator.

An **append(symbol)** vertex has indegree 1 and outdegree 1. Let the multiset assigned to the incoming edge be M' . Then the multiset associated with the outgoing edge is obtained by appending **symbol** to the right end of each string of M' .

A **separate** vertex has indegree 1 and outdegree 2. One of the outgoing edges is labeled by a **symbol**. Let M' be the multiset associated with the incoming edge. The subset of M' that contains **symbol** is associated with the outgoing edge that is labeled **symbol**, and the test tube of the remaining strings is associated with the other edge. If an edge is labeled with weight p , then the number of strings in the multiset associated with that edge is $p|M'|$; otherwise the graph is not a valid generator.

A **merge** vertex has outdegree 1 and indegree at least 2. The multiset associated with the outgoing edge is simply the union of the multisets of all incoming edges. A **no-op** vertex has indegree 1 and outdegree 1. The test tube associated with the outgoing edge is simply the test tube associated with the incoming edge. The **no-op** operation is included simply to make it easy to organize the graph into levels.

The multiset of strings associated with the edge entering the sink vertex is the solution set (possibly multiset) generated by the generator. The size of this set is the same as the size of the multiset associated with the edge leaving the root.

Examples. Lipton proposed a generator (in the case $k = 2$) to generate a solution set representing all k^n strings over an alphabet of size k . The strings in the solution space are of the form $b_1b_2 \dots b_n$ where $b_i \in \{1_i, 2_i, \dots, k_i\}$. The generator consists of a sequence of n split nodes of outdegree k , where the j th edge from the i th split node leads to an **append(j_i)** node, followed by a k -way merge. We call this a k^n -generator.

In Figure 1, we present a generator that generates a solution set representing the set of all binary strings of length n with exactly k 1's. We call this a $\binom{n}{k}$ -generator.

2.2 A DNA Model for Languages

Let L be a problem in NP. A *DNA algorithm* for L is a polynomial time algorithm (for a standard computer) with the following properties. Given an instance x of L , it outputs a (valid) generator for x and a specification of a DNA computation phase. The DNA computation phase may be specified by a dag similar to a generator, except that there are no weights associated with the edges, vertices need not be organized into levels, there are no vertices labeled with the operations **split** or **no-op**, and the last (sink) vertex is labeled with **test-if-empty**. The test tube labeling the edge from the root is the output of the generator; test tubes labeling the other edges are the result of the operations on the vertices. The test tube labeling the edge into the sink is not empty if and only if the instance is in L .

There are several resources of a DNA algorithm that are important to measure, as a function of input size. The primary ones that we consider are: (i) the solution space size, (ii) the total length of the computation, measured as the length of the

path from root to sink in the generator, plus the maximum length of a path from root to sink in the dag specifying the DNA computation, (iii) the number of operations performed during the solution space generation phase and the computation phase, and (iv) the number of test tubes needed. Another resource that we will introduce later is the *redundancy* needed due to the probabilistic nature of the low-level **split** operation.

Algorithms for 3Sat, 3-Coloring and other problems can be found in [2, 8]. The 3SAT algorithm has a solution space of size 2^n and requires $\Theta(n + m)$ operations, where n is the number of variables and m is the number of clauses. The 3-Coloring algorithm has a solution space of size 3^n and requires $\Theta(n + m)$ operations, where n is the number of nodes and m is the number of edges. An algorithm for the Circuit Sat problem can be found in [4]. This problem is to determine if an n -input circuit (with “and”, “or” and “not” gates and one designated output gate) has an input that sets the output to 1. Their algorithm has a solution space of size 2^n on an instance circuit with n inputs. The output of the solution space generator is simply the test tube consisting of all 2^n possible inputs to the circuit. Briefly, in the DNA computation, the gates of the circuit are processed in topological order. When processing gate g , inputs for which g evaluates to 1 are separated from those for which g evaluates to 0. The appropriate value of g is appended to each input (in parallel for each of the two possible values of g), and the two test tubes are merged. In this way, the value of g is available when processing other gates that have the output of g as input. In this algorithm, the size of each element of the solution space is initially of length n but increases to length $n + m$, where m is the number of gates in the circuit.

More generally, if a test tube contains some subset of the 2^n possible inputs to a circuit, the same algorithm can be used to determine if any element of this subset sets the output of the circuit to 1. Furthermore, if the circuit has several outputs, then the same algorithm transforms the

initial test tube into a test tube in which each possible input to the circuit has appended to it the value of every output gate.

3 New Algorithms

We present four algorithms that exploit our DNA computing model in different ways. Each algorithm is described by the size of its solution space, since this is the most expensive resource and thus the one that is most important to minimize. In what follows, we assume that the input to each algorithm is a connected, undirected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. We summarize the resources used by each algorithm in Table 1.

A $n1.89^n$ 3-Coloring Algorithm. The generator produces a test tube T of strings that represent all subsets of V of size $\leq \frac{n}{3}$. Each string in the solution space is of the form $b_1b_2 \dots b_n$ where $b_i \in \{0_i, 1_i\}$; this string represents the subset of vertices i of V for which $b_i = 1_i$. Roughly, this can be done by a generator that is formed by merging the results of the generators from Figure 1 that generate strings with exactly $1, 2, \dots, k$ 1's. The generator is of size $O(n^3)$ and has length n . T contains $\sum_{j=1}^{\frac{n}{3}} \binom{n}{j} \leq n1.89^n$ elements.

In the computation phase, the elements of T that are not independent subsets of V are first discarded. To do this, for every edge e of G , those elements of the solution space containing both endpoints of e are **separated** from those containing one or none of e 's endpoints, and discarded. The elements remaining in T are independent subsets of V of size at most $n/3$. Then, for each remaining element s of T , the algorithm tests if the subgraph of G induced by the set $V - s$ is bipartite. This can be done in $O(n^2 + m)$ time (details omitted).

A 1.67^n Independent Set Algorithm. Briefly, this algorithm uses a preprocessing phase

Problem	Solution Space Size	Redundancy	Length of Computation	Number of Operations	Number of Test Tubes
3-Coloring	$n1.89^n$	$O(n^7)$	$\Theta(n^2 + m)$	$\Theta(n^2 + m^2)$	$\Theta(n)$
	2^n	$\Theta(n)$	$\Theta(m)$	$\Theta(m)$	$\Theta(1)$
Ind Set	1.67^n	$\Theta(n)$	$\Theta(m)$	$\Theta(m)$	$\Theta(n)$
	1.51^n	$O(n^7)$	$O(n^2m^2)$	$O(n^2m^2)$	$\Theta(n^2)$

Table 1: Asymptotic bounds for resources required by the four DNA algorithms.

to identify small, well-connected components in G . The solution space is then constructed to represent the set of subsets of the vertices of the graph that contain no adjacent pair of vertices from the same component.

The generator first selects a maximal set S of vertex-disjoint subgraphs from G , with exactly 4 vertices and at least 4 edges. These are called 4-components. A case enumeration of the possibilities shows that each such 4-component contains at most 7 distinct independent sets out of the 16 possibilities. From the remaining vertices of G , a maximal set of vertex-disjoint subgraphs that are connected and have exactly 5 vertices are selected and these 5-components are added to S . Any such 5-component contains at most 13 independent sets. Let V' be the set of vertices in the components of S . Note that there may still be vertices in $G - S$.

The solution space consists of strings of the form $b_1b_2 \dots b_{|V'|}$, where $b_i \in \{0_i, 1_i\}$. This string represents the subset of vertices $i \in V'$ for which $b_i = 1_i$. Moreover, no subset (string) in the solution space contains a pair of vertices that are neighbors in a 4-component or a 5-component of S . Initially, the solution space consists of a multiset T of empty sets. For each 4-component in turn, a 7-way **split** is performed on T . A different combination of 4 1_i 's and 0_i 's are **appended** to each of the 7 resulting test tubes, corresponding to the 7 different independent sets contained in the 4-component. Then, the 7 sets are merged back together to reform T . 5-components are handled in an analogous manner, except that the solution space is **split** 13 ways and 5 ver-

ties are **appended** to each of the 13 test tubes. It is not hard to see that the worst-case size of the solution set occurs when all components in S are 5-components, and that the number of 5-components is at most $n/5$. Therefore, the worst-case size of the solution space is at most $13^{n/5} = 1.67^n$.

The computation phase proceeds as follows. First, for every edge e connecting vertices in two different components of S , those elements of the solution space containing both endpoints are **separated** from those containing one or none of e 's endpoints, and discarded. The elements remaining in T are exactly the independent subsets of V' .

Next, the vertices in $G - S$ are handled. The goal is to add as many of these remaining vertices as possible to each element of the solution space while ensuring that each element remains an independent set. A vertex i can be "added" to the solution set by first **separating** out all independent sets which contain a neighbor of i , and **appending** 0_i to these strings. Then 1_i is **appended** to the remaining strings. This ensures that the elements of T remain independent subsets of G , but the vertices of $G - S$ must be "added" in an order that guarantees that the maximum independent set will be in the solution space. This can be done by exploiting the fact that every connected component in $G - S$ is either a triangle or a tree.

Any triangle in $G - S$ is not connected to any other vertex in $G - S$, because otherwise that 4-component could have been added to S . Since at most one of the 3 vertices in a triangle can be in an independent set, they can be added in an

arbitrary order. For each tree in $G - S$, a vertex is chosen arbitrarily to be the root of the tree. Then, vertices are added in postfix order, which ensures that the children of a vertex are added before that vertex itself. This order ensures that the maximum number of vertices of the tree are being added.

Finally, the algorithm determines whether an element of the resulting test tube T is of cardinality at least k . To do this, T is partitioned into test tubes T_i , $0 \leq i \leq k$, such that for $i < k$, each T_i contains all solutions of size i and T_k contains all solutions of size at least k . Initially, $T_0 = T$ and the remaining test tubes are empty. Then, all solutions containing vertex 1 are **separated** from T_0 and **merged** with T_1 . More generally, for every vertex i , for every test tube T_j , all solutions containing t_i are **separated** from T_j and **merged** with T_{j+1} . After this is done for all vertices i , all elements of the solution space with at least k vertices are in T_k . A **test-if-empty** operation performed on T_k determines whether G has an independent set of size at least k .

A 2^n 3-Coloring Algorithm. This algorithm is the first to use the **separate** operation in the solution space generation phase. The DNA algorithm first selects a rooted spanning tree of G , and fixes the color of the root (vertex 1). The solution space consists of strings of the form $c_1 c_2 \dots c_n$ where each $c_i \in \{b_i, r_i, g_i\}$, representing the fact that vertex i can have color blue, red or green. Moreover, in any string, c_1 is the fixed root color, and if i is j 's parent in the tree, then j cannot have the same color as i . This solution space is constructed as follows. Initially, the solution space consists of a test tube T of empty strings. The vertices of the spanning tree are considered in prefix order. When considering vertex i , T is **separated** into 3 different tubes, say B, R and G , based on the color of i 's parent, j , in the tree. Then, test tube B is split in 2, r_i is **appended** to one and g_i to the other. Test tubes R and G are handled similarly. Finally, all 6 test tubes are **merged** to reform T .

The computation phase then simply selects out those solutions which are not valid colorings by examining each edge of E that is not in the spanning tree in turn, and discarding strings in which both endpoints are the same color.

A 1.51^n Independent Set Algorithm. Briefly, this algorithm tries to find two disjoint independent sets of size $n/7$ in G , which induce a bipartite subgraph B . After generating a solution space of independent sets of $G - B$, the vertices of B are added later, without increasing the size of the solution space. Although we obtain an improved solution space size, the required computation includes simulation of a circuit for bipartite matching. We assume that the problem is to find an independent set of size $k \geq 2n/7$; otherwise, we can solve the problem with an abbreviated version of our algorithm which has an even smaller solution space.

The DNA computation first determines if there is an independent set of size $n/7$. The generator of Figure 1 is used to generate all subsets of V of size $n/7$. The solution space is of size at most 1.51^n . Then, the computation phase discards all those subsets that are not independent (as in previous algorithms), and a **test-if-empty** operation is performed on the test tube of remaining strings. If there is an independent set I of size $n/7$ in this test tube, our algorithm needs to compute I in order to proceed. This could be done by extending our model to include a "decode" operation that returns the value of a string in a test tube. We can still compute I without this operation, but in a very brute force fashion that requires the solution space generation and computation phases to be repeated $n/7$ times, with one vertex of I being computed each time. In the same way, the algorithm tries to find another independent set J of size $n/7$ in the graph G with the subgraph induced by I removed. The solution space for this computation is of size at most 1.47^n . If no independent set is found in either of the first two steps, then we know that there are no independent sets of size $k > 2n/7$. Otherwise,

the algorithm continues as follows.

Next, a solution space containing all independent sets of $G - B$ is computed. Let B be the bipartite subgraph of G induced by $I \cup J$. Let P be a maximal set of disjoint pairs of adjacent vertices from $G - B$ and let S be the independent set of vertices of $G - B$ that are not in P . We can bound the size of S from above by $n/7$, by simply swapping I and S if S exceeds $n/7$ vertices. The solution space consists of subsets of $V - I - J$ that contain no pair of P . Note that each pair of vertices in P contains 3 independent sets out of the four possibilities. Hence, the size of this solution space is $2^{|S|}3^{|P|}$. Obviously, the worst-case solution space results when $|S|$ is maximized at $n/7$. With $n/7$ of G 's vertices in S , at least $n/7$ in I , and $n/7$ in J , at most $4n/7$ vertices ($2n/7$ pairs) remain in P . Hence, the worst-case solution space size is $3^{2n/7} \times 2^{n/7} = 1.51^n$. Let T be the resulting test tube. All elements of T that are not independent sets are discarded.

Finally, the vertices of B are added to the elements of T in the following way. For every vertex i in B , **append** 1_i to those elements of T which do not contain a neighbor of i from the set $V - I - J$, and 0_i to the remaining sets. The resulting test tube T contains strings that represent sets of vertices that induce bipartite subgraphs of G . Moreover, each maximum independent set of G is a subset of at least one of these sets. The algorithm must now find the maximum independent sets of the bipartite graphs of T . We observe that this problem can be solved with a circuit of size $O(n^2m^2)$ (using bipartite matching; see [7]). The circuit has n inputs representing the presence or absence of each vertex of G in a bipartite graph, and n outputs which indicate the maximum independent set. The circuit first finds a maximal matching M in the bipartite graph. Then, for each of the $O(n)$ free nodes (i.e. nodes not incident to an edge in M), the circuit performs a breadth-first search along alternating paths beginning at the free node. The search is simulated by repeatedly expanding sets of edges along disjoint alternating paths. After

all eligible edges have been covered, the circuit searches through the sets of edges to find an edge which completes an augmenting path (an alternating path with a free node at each end). If an alternating path is found, it can be "inverted" to increase the size of the current matching by one. Once these three steps (each requiring $O(nm^2)$ gates) have been performed on every free node, the resulting matching is maximum. A maximum independent set can then be determined from this matching by selecting any remaining free nodes, as well as one node from each edge in the matching. One node from every edge is guaranteed to be in a maximum independent set; otherwise, we would contradict the assumption that the matching is optimal. A DNA computation can simulate this circuit, resulting in the n outputs being appended to each element of T .

All that remains is to isolate those outputs of size at least k . This process is identical to the final step of the 1.67^n independent set algorithm.

4 Probabilistic Implementation of Split Operation

Suppose that in a generator, the **split** vertex is replaced by the following **probabilistic-split** vertex. Suppose that the edges leaving this vertex are weighted p_1, p_2, \dots, p_k . Then, the operation, applied to a test tube with N strings, produces a partition of the test tube into k test tubes containing p_1N, p_2N, \dots, p_kN , strings, respectively. Moreover, each possible such partition is equally likely. In this section, for several graph types, we describe the redundancy that is then needed in the solution space.

We first consider the n -vertex k^n -generator of Lipton. Let $p = 1/k$ and let $M = 1/p^n$. Our goal is to find bounds on the number N of strings needed in the initial test tube in order that all M distinct strings are generated with high probability. The distribution of strings produced by **probabilistic-split** equals the distribution produced by the following process. Suppose that

instead of generating all strings in parallel, one string at a time is generated. The generation of a string is called a trial. The string generated at each trial is determined by a path chosen randomly in the graph in the following way. Associated with each **split** vertex is an urn, which initially has N balls of k different colors, with exactly pN of each color. At each trial, a ball is removed from every urn. The color of the ball determines which edge is in the path.

In the first trial, each of the possible M strings are equally likely. If the same were true in every trial, that is, if each of the M strings is equally likely, then solutions to the coupon collector's problem show that $N = \Theta(M \log M)$ trials are necessary and sufficient, in order that all M distinct strings are generated with high probability. However, in our urn model, the probability of generating a particular string in a given trial depends on the strings generated in previous trials. We show in Theorem 1 that it is still the case that $N = \Theta(M \log M)$ trials are needed to generate all strings with high probability. First, we introduce some notation and state a key lemma. The details of the proof are omitted.

Let red be one of the colors used in the urns. Let X_t be the number of red balls after t samples from the urn, so that $X_0 = pN$, $X_1 = pN$ with probability $1 - p$ or $pN - 1$ with probability p , and so on. The key lemma is:

Lemma 1 *For any $\epsilon > 0$, with probability at least $1 - N \exp(-p^2 N^{2\epsilon}/16)$,*

$$|X_t - E[X_t]| < pN^{1/2+\epsilon}, 1 \leq t \leq N/2.$$

This lemma shows that in the first $N/2$ trials, the choice of ball from an urn is relatively unbiased, with high probability. As a result, each of the first $N/2$ strings is chosen in an almost unbiased manner, and so N is as predicted by the coupon collector's problem. This is made precise in the following lemma, whose proof is straightforward.

Theorem 1 *To obtain exponentially small error, a redundancy of $\Theta(n)$ is sufficient for the k^n -generator.*

The generator for the 1.67^n Independent-Set algorithm is very similar to a k^n -generator, except the outdegrees of the **split** vertices may not all be equal. A bound is obtained simply by letting $k_{max} = 13$ be the maximum outdegree of a vertex, $p_{min} = 1/k_{max}$ and using these values in the proof of Theorem 1. Thus, a redundancy of $\Theta(n)$ is necessary and sufficient to guarantee that the desired solution space is generated with all but exponentially small probability.

In the 2^n 3-Coloring algorithm, the **split** operations are applied to test tubes with $N/3$ strings, rather than N strings. Moreover, the number of **split** vertices for an n -vertex graph is $3n$. The same analysis as in Theorem 1 can be applied with these changes, and the fact that $p = 1/3$, to see that the redundancy is again $\Theta(n)$.

Finally, we have an upper bound of n^7 on the redundancy for the $\binom{n}{n/k}$ generator that is used in our $n1.89^n$ 3-Coloring and 1.51^n Independent Set algorithms. We are working on improving this bound. We have done simulations of our $\binom{n}{n/2}$ -generator for $n \leq 22$, and in all simulations a redundancy of $2n$ is sufficient for 0 error.

5 Conclusions

In this paper, we have described algorithms for 3-Coloring and Independent Set that use only the restricted set of operations proposed by Adleman, while using a solution space of only $n1.89^n$ and 1.51^n , respectively. These algorithms represent a big improvement over naive exhaustive search. If we again assume Adleman's speculation that a solution space of size 2^{70} is feasible, we can solve instances of 3-Coloring and Independent Set with 70 and 118 vertices respectively, in contrast with the limits of 44 and 70, respectively, for the naive algorithms.

These improvements are not sufficient to claim that DNA computation is practical for such problems. However, there is no reason to assume that they are optimal. Improved algorithms may be

possible on our model, or if new operations are added to the model. We note that heuristic algorithms may also be useful in the DNA computing model.

Our algorithms are obtained by revising Lipton's model so that the solution space is pruned and tailored to the problem instance. Efficient methods for generating solution spaces other than the set of combinatorial possibilities are interesting in their own right. Fodor *et al.* [5] and Pease *et al.* [6] described a method they used for generating sets of DNA strands that can be described using "polynomial notation." For example $(A + C)(G + T)$ describes the set $\{AG, CG, AT, CT\}$. Just as circuits can be used to represent functions more efficiently than boolean formulas, it appears that their method may generalize to allow efficient generation of even richer sets of oligos than those describable by polynomial notation.

Finally, we present the first analysis of errors in the solution space generation phase due to splitting of the contents of test tubes. This is a very basic operation, so our analysis is likely to be useful in any potential application of DNA computation. We show how to generate the set of binary strings of length n with exactly k 1's, using $O(nk)$ operations and a redundancy of at most n^7 . We would like to improve this upper bound on the redundancy. An alternative method of generating this set of strings was proposed by Dan Boneh (personal communication), using an additional operation that separates strings according to their length (see [4] for a description of this operation).

References

- [1] L. M. Adleman, *Molecular Computation of Solutions to Combinatorial Problems*, Science, **266**, 11 November, 1994, pp. 1021–1024.
- [2] L. M. Adleman, *On Constructing a Molecular Computer*, Manuscript, Department of Computer Science, University of Southern California, 1995.
- [3] R. Biegel and D. Eppstein, *3-Coloring in Time $O(1.3446^n)$: a No-MIS Algorithm*, Proc. 36th

Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1995, pp. 444–453.

- [4] D. Boneh, D. Dunworth, R.J. Lipton and J. Sgall, *On the Computational Power of DNA*, Technical Report Number TR-499-95, Computer Science Department, Princeton University, Princeton, New Jersey, 1995.
- [5] S. P. Fodor, J. L. Read, M. C. Pirrung, L. Stryer, A. T. Lu and D. Solas, *Light-Directed, Spatially Addressable Parallel Chemical Synthesis*, Science **251**, 1991, pp. 767-773.
- [6] A. C. Pease, D. Solas, E. J. Sullivan, M. T. Cronin, C. P. Holmes and S. P. Fodor, *Light-Directed oligonucleotide arrays for rapid DNA sequence analysis*, Proc. Natl. Acad. Sci. USA, **91**, May 1994, pp. 5022–5026.
- [7] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA.
- [8] R.J. Lipton, *DNA Solution of Hard Combinatorial Problems*, Science, Vol. 268, 28 April 1995, pp 542–548.
- [9] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
- [10] I. Schiermeyer, *Solving 3-satisfiability in less than 1.579^n steps*, Proc. 6th Workshop on Computer Science Logic, Springer-Verlag, 1993, pp. 379-394.

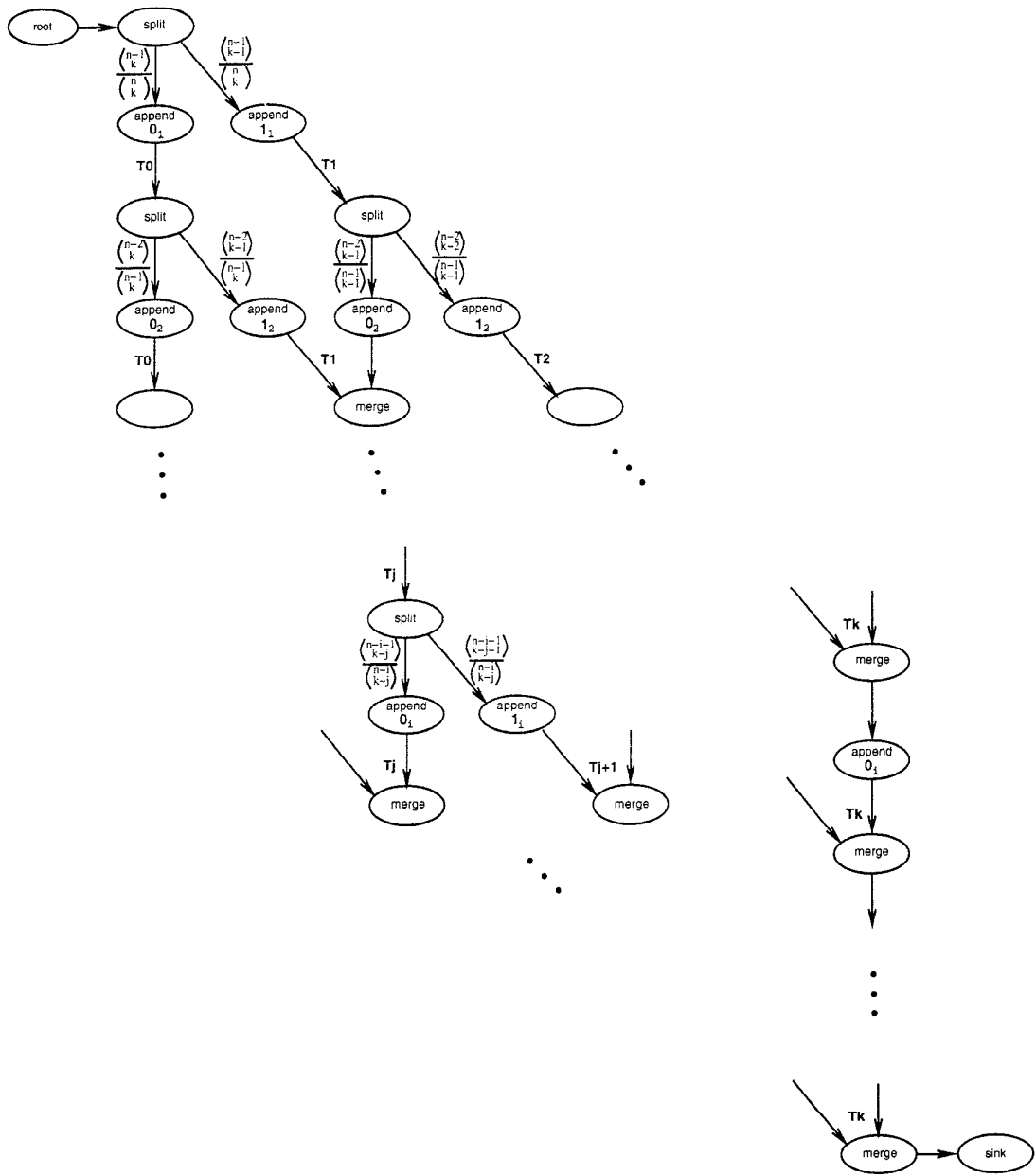


Figure 1: An n choose k generator. The test tube labeling the edge from the root contains all empty strings, and the test tube labeling the edge to the sink contains the desired solution space. Each test tube T_i , $1 \leq i \leq k$ stores strings with i 1's. Initially, T_0 consists of n choose k empty strings. At the i th stage, for each j , $1 \leq j < k$, a weighted **split** is performed on T_j , so that $\frac{k-j}{n-i}$ of T_j 's elements are in one tube and $\frac{n-i-(k-j)}{n-i}$ in the other. We **append** 1_{i+1} to the strings in the first tube and **merge** it with T_{j+1} . To the second tube we **append** 0_{i+1} ; and **merge** it with T_j . T_1 and T_k are handled somewhat differently than the general case, as shown.