

Sensitivity Analysis on Bio-op Errors in DNA Computing

Daniel Bilar, *Member, IEEE*

Abstract—We simulate the biological operations of a DNA computational algorithm for a combinatorial problem. We then perform sensitivity analysis in which we vary bio-op error rates to see which bio-ops affect the end result. Finally, we review three approaches to tune the algorithm in order to minimize significant error.



1 INTRODUCTION

THE use of biological computing has shown promise in finding solutions to certain types of combinatorial problems, such as the Hamiltonian Path problem, DES code breaking and knapsack problems [1], [2], [3]. In addition, several researchers have shown that it is theoretically and practically possible to build *universally programmable* DNA computers [4], [5]. For an overview of DNA computing concepts and terms, the reader is referred to [6]. One of the practical difficulties that arise in implementing DNA computing algorithms is controlling the error rate at each computational step. Unlike their silica-discrete counterparts, biological operations (bio-ops) produce incorrect results from time to time. The error rates typically range from 10^{-5} to 0.05 [7]. As bio-engineers improve their techniques, these rates will indubitably fall. One could call this method “improvement through refinement”.

One could with some empirical justification maintain that the error rate difference between biological and silicon-based computation is one of kind, not of degree: Biological computations are inherently probabilistic, unlike logical computations. Hence, “improvement through renovation”, that is, reformulation of the problem in terms of probabilistic algorithms, might yield some interesting insights.

Each bio-op produces a correct result within the limits of some type I or type II error. Then, one of the goals of DNA algorithms would be to attempt the error of the final result by taking into account the error rates of the various bio-operation stages.

The paper is organized as follows: We simulate the biological operations of a DNA computational algorithm for the Shortest Common Superstring problem. We then perform factorial experiments for a sensitivity analysis. We vary bio-op error rates to see whether some bio-ops

affect the end result more than others. Lastly, we review three ways to tune the algorithm in order to minimize significant error. The appendices present the MATLAB simulation code and the experimental data.

2 SHORTEST COMMON SUPERSTRING

2.1 Description

The problem chosen for the sensitivity analysis is the “Shortest Common Superstring Problem”. We are given an alphabet Σ , a finite set R of strings from Σ^* (the set of all words over Σ) and a positive integer K . Find a string $w \in \Sigma^*$ with length $|w| \leq K$ such that each string $x \in R$ is a substring of w .

As an NP-complete combinatorial problem, it is not known whether a poly-time algorithm on a conventional silicon computer can be found for non-trivial setups. A DNA-based computer’s ability to comb in parallel through the solution space and sift out a solution (if any) may be able to break that polynomial time barrier.

We adapt Gloor’s algorithm for our simulating the DNA-computing combinatorial approach [8],

- 1) Encode all the strings $x_1, x_2, \dots, x_n \in R$ as DNA strands.
- 2) Generate all possible DNA strands w of length less than or equal to K . This is the solution population
- 3) Let x_j be a string of R . From our solution population, select only the ones which contain x_j as a sub-string. Let this be our new solution population. Repeat this step for each string $x_i \in R, 1 \leq i \leq n$
- 4) If our solution population is non-empty, return ‘Yes’ and the solution string(s). Otherwise, return ‘No’

These bio-operations used in every step have error rates several orders of magnitude greater than their logical counterparts in electronic computing and thus can greatly affect the outcome of the algorithm. Table 1 lists the error rates associated with each step. Simulation of algorithm was done in MATLAB, with environmental parameters chosen to ensure both manageable run-time

• Department of Computer Science, University of New Orleans, USA
E-mail: dbilar@cs.uno.edu
• Daniel Bilar is with the University of New Orleans

TABLE 1
Error rates of bio-operations [7][9]

Step	Bio-op	Type I Error (rate)	type II Error (rate)
1) Encoding sub-strings	Synthesizing through sequential coupling	NA	Wrong letter is bonded (0.05)
2) Generate solution population	Synthesizing through sequential coupling	NA	Wrong letter is bonded (0.05)
3) Match sub-strings to solution population	Extraction using affinity purification	Correct match is not recognized as match (0.05)	Incorrect match is recognized as match (10^{-6})
4) Detect and output final solution	Sequencing using polymerase chain reaction and gel electrophoresis	Correct match is not recognized as match (0.05)	Incorrect match is recognized as match (10^{-5})

TABLE 2
All possible solutions of length $K \leq 6$ and sub-string matches gg, t, cg, tg, tgg

Solution #	String
1	tggcg
2	cgtgg
3	tggcgc
4	cgtggc
5	tggacg
6	tggccg
7	tgggcg
8	atggcg
9	ctggcg
10	gtggcg
11	ttggcg
12	tgttcg
13	tggcgg
14	cgtggg
15	cgatgg
16	cgctgg
17	acgtgg
18	ccgtgg
19	gcgtgg
20	tcgtgg
21	cggctg
22	cgttgg
23	tggcgt
24	cgttgt
25	tggcga
26	cgtgga

for one search on an old Intel 80386 platform and a sufficiently large final solution population given perfect error conditions. As such, five sub-strings - gg, t, cg, tg, tgg - of length three or smaller in R were specified and $K \leq 6$ chosen as the solution length bound. These values were set to insure a manageable run-time, i.e. under thirty minutes, for one search and a sufficiently large final solution population given perfect error conditions. Table 2 lists this reference solution against which results affected by type I and II errors were compared.

2.2 Simulation

After gauging a few trial runs, a factorial experiment regime was performed in which we varied error rates for three bio-ops delineated in Table 1: [Step 1, type II] with 2 levels, [Step 2, type II] with four levels and [Step 3, type I] with four levels for a total of 32 experiments, repeated five times with different random number generator seeds, giving 160 individual runs. The levels are listed in Table 3. The error rates were adjusted by one order of magnitude until they were insignificant relative to the problem. Errors that were not investigated failed to occur with sufficient empirical frequency to warrant inclusion in a sensitivity analysis at this time. The results for the averaged 32 experiments are listed in the appendix B.

3 SENSITIVITY ANALYSIS

It was found that the hit rate was most sensitive to the type II errors in step 1. Since step 1 encodes the sub-strings, this result is plausible: An error at that step makes the algorithm search for the wrong sub-strings. Keeping the error in step 1 constant, we found that the error rate of step 2 has virtually no effect on the hit rate. This is reasonable, as well, since the probability of an error affecting one of the reference solution encodings in all possible encodings is quite small. The type I error of step 3, however, in conjunction with a lower [Step 1, type II] error, pushed the hit rate above the 90% mark. The extraction process, on the other hand, is of vital importance, since it serves to match the search strings with the super-string. A lower error rate at this step is prone to positively affect the end results provided the search strings are properly encoded.

4 TUNING THE ERRORS: THREE APPROACHES

4.1 Good Encoding

As the sensitivity analysis has shown, the correct results are most sensitive to the errors that occur in step 1, namely false encoding of the search strings. The primary practical mechanism that produces this error is *hybridization stringency*. Hybridization stringency

TABLE 3
Bio-op error levels for 3-factor experiments

Step	Type I Error Levels	Type II Error Levels
1) Encoding sub-strings	NA	0.05, 0.005
2) Generate solution population	NA	0.05, 0.005, 0.0005, 0.00005
3) Match sub-strings to solution population	0.05, 0.005, 0.0005, 0.00005	NA

refers to the number of complementary base pairs that have to match for DNA oligonucleotides to bond. A good encoding scheme should prevent false matching of oligonucleotides, such as A's bonding with G's. Deaton et al analyzed the encoding of vertices of a Hamiltonian Path Problem and found the upper bound on the number of vertices that can be encoded in oligonucleotides of length n without producing mismatches given by (1)

$$|C| \sum_{i=0}^t \binom{\frac{n}{2}}{i} (q-1) \leq q^{\frac{n}{2}} \quad (1)$$

where t is the number of errors that occur in hybridization, q is cardinality of the alphabet ($q = 4$ for DNA), and $|C|$ is the number of vertices [10]. Subsequently, they used a genetic algorithm to search for mismatch-free encoding for the vertices of the path; mismatch-free defined as every codeword being a distance greater than t from any other codeword. If the Hamming bound is satisfied, no type II matching errors will occur.

This encoding scheme is the biological pendant of the Hamming error-correcting code used in conventional computing for detection and correction of bit errors. The principal difference is that, unlike the Hamming code which only catches single or double errors, the number of errors that can be corrected subject to the Hamming bound is unrestricted which is fortunate given the much higher error rate of bio-ops. On the other, the oligonucleotide encoding has to be mismatch-free as defined above to satisfy the Hamming bound, which for a given problem may not be possible. The Hamming code does not place a condition on the encoding of the bits; it will catch errors in arbitrary 4-bit patterns. Thus, the added error flexibility has to be bought with stringent, non-random oligonucleotide encoding.

4.2 Multiplexing

Whereas good encoding is a method of making an input as error-resistant as possible, an complimentary approach would be to have a system which - assuming a certain number of faulty inputs - can 'rebound' from the error. v Neumann investigated the properties of such a scheme for general automata [11]. The method is as follows: Imagine many inputs, all carrying the same message to a black box. Given an input error rate and an operation error rate, a critical level of these inputs is determined for a desired output error rate. A group

of inputs that is higher than the critical state will be interpreted as a positive state. A group of inputs that is lower than the critical state will be interpreted as a negative state. v. Neumann referred to this method as multiplexing.

Reformulated for the DNA computing domain, this proposal amounts to the following: For every bio-op with error rate ϵ , fix your output error rate ψ to a desirable level by replicating the inputs N times. Given N , find your critical level δ . Now, if at least the fraction $1 - \delta$ of your inputs remains the same, the operation produces a positive result, i.e. it succeeds in achieving an error rate no larger than ψ . Conversely, if at most the fraction δ of your inputs is the same, the operation produces a negative result, i.e. it has failed to achieve an error rate smaller than ϕ . The interval zone $(\delta, 1 - \delta)$ is one of uncertainty, where the error rate may or may not have been achieved.

The drawback of this approach is that N becomes very large as we aim to decrease the probability of uncertainty. N can be deduced from the error integral (2). The dependency is illustrated in Table 4 with bio-op error rate $\epsilon = 0.005$. We see that in order to achieve a reasonable uncertainty probability of 10^{-6} , we would have to multiplex every input to every bio-operation around five thousand times. This in itself is not unreasonable; the real limitation of this 'rebounding' scheme is its inability to handle feedback well. There are situations where, far from canceling the errors, multiplexing will actually amplify them. This limitation is not crucial for combinatorial problems (such as the Shortest Common Superstring under investigation), but it is critical for problems, which are traditionally tackled with divide-and-conquer algorithms. Thus, multiplexing in DNA computing helps to stabilize errors in algorithms with little to no data dependencies. The errors of more intricate algorithms which exhibit a high degree of data dependency may not be stabilized and might actually be amplified. This suggests that algorithms might have to be reformulated to suit the particularities of DNA computing.

$$\rho(N) = \frac{1}{\sqrt{2\pi k}} e^{-\frac{k}{2}}, \text{ with } k = 0.62\sqrt{N} \quad (2)$$

4.3 Constant Volume Transformation

The two methods we have seen so far concentrated on making improvements of the individual operations,

TABLE 4
Probability of uncertainty as a function of N

N	1000	2000	3000	5000	10000	20000
$\rho(N)$	$2.7 * 10^{-2}$	$2.6 * 10^{-3}$	$2.5 * 10^{-4}$	$4 * 10^{-6}$	$1.6 * 10^{-10}$	$2.8 * 10^{-19}$

either by improving the operand, or by statistically improving the error rate of the operators. Boneh, taking a broader view on DNA algorithms, tries to adapt the algorithm to the particularities of DNA computing [9]. He classifies DNA algorithms as ‘Decreasing Volume’ if the number of strings decrease as the algorithm executes, ‘Constant Volume’ if the number remains the same and ‘Mixed’ for algorithms which fit neither of the previous classifications. Hence, the problem under investigation falls under the ‘Decreasing Volume’ rubric, since our solution population decreases which each execution of step 3. Hence, Boneh’s approach consists of transforming an algorithm which is decreasing volume into one that is constant volume. In our example, this entails modifications to bio-op steps three and four in Table 1.

(3*) Let s be the number of extraction steps, and let the initial solution population be 2^n strings. Double the solution population every $\frac{s}{n}$ steps using a PCR (a DNA amplification technique) operation.

(4*) Pick m strands at random from the final solution population and check whether at least one of them is the desired solution. If not, report failure.

Following [9], assume the worst case that there is only one desired solution in the initial population of 2^n . Let P_s be the probability that a solution managed to survive extraction and is in the final population, and let P_r be the selection probability, i.e. the probability that at least fraction f in the final population represent solutions. Then the probability of the algorithm succeeding, i.e. of finding a solution, is bounded below by 3

$$P_s P_r (1 - f)^m \quad (3)$$

The crucial step consists of bounding P_s by the modification in (3*). Recall that every s/n steps, the solution population is doubled. Hence, through this quasi-exponential growth process every s/n steps, chances increases that some solution string will survive all extractions. The probability P_s of this event is given by (4)

$$2 - \alpha^{-\frac{s}{n}}, \text{ with } \alpha \text{ being the type I error} \quad (4)$$

There are fine points that should be stressed. This approach assumes that the Polymerase Chain Reaction (PCR) operation is error-free; this can be accommodated

by further reducing α . Also, algorithms that are constant-volume to begin with are not susceptible to this error-reducing technique, since the quasi-exponential growth would soon make the bio-mass unmanageable.

5 CONCLUSION

We have seen that the brute-force DNA computing solution to the Shortest Common Superstring problem is most sensitive to [Step 1, type II] and [Step 3, type I] errors. We have outlined several approaches that have been proposed to minimize the error in the final solution. *Good Encoding* makes the input as error-resistant as possible. *Multiplexing* tried to make the operation as error-resistant as possible. *Constant Volume Transformation* tries to make an algorithm as a whole as error-resistant as possible. Aesthetically, we find the latter approach most appealing. By accepting the basic premise of DNA computing - namely, that operations are inherently probabilistic - it strives to adapt the algorithm accordingly. Each distinct computing environment may call for particular algorithmic finetuning, be they digital, DNA, quantum or other computing environments.

APPENDIX A MATLAB CODE

Listing 1. DNABioOpSensitivity.m

```
% Simulation of brute force solution
% of greatest common superstring problem
% with varying error rates

function [result_length, exp_num_hits, exp_num_misses, coords, ...
        final_match, result, hit, matches, ...
        x, w, num_type1, num_type2] = DNABioOpSensitivity();

global result_length exp_num_hits exp_num_misses coords final_match
...
result hit matches x w num_type1 num_type2;
%% clean up
close all; clear all;

% environmental variables
t0=clock;
seed = [9999 12663 18765 23 88888]; % random number generator seed
num_seed = length(seed);
filename = 'ref'; %file containing reference solution with no errors

% program vars
alphabet = ['a' 'c' 'g' 't'];
card_alpha = length(alphabet); %number of elements in alphabet
num_substring = 5; %number of sub-strings to be generated
x = {'gg', 't', 'cg', 'tg', 'tgg'}; %initializing the sub-strings
substring_length = 3; %maximum length of a sub-string
K = 6; %solution string's maximum length
max_sub_len = 0; %maximum sub-string length
coords = []; %row, column coordinates of final solutions
ref = []; %reference solution
exp_num_hits = zeros(1, num_seed); % the number of correct solutions found
exp_num_misses = zeros(1, num_seed); % the number of incorrect solutions found
result_length = zeros(1, num_seed); % the number of solutions found
algorithm_steps = 4; % These are the 4 steps of the DNA algorithm
type1 = [0 0.005 0.05]; % Type I error probability of individual steps
type2 = [0.05 0.05 1e-6 1e-5]; % Type II error probability of individual steps
num_type1 = repmat(0, num_seed, algorithm_steps); num_type2 =
repmat(0, num_seed, algorithm_steps);

% for the parameters above, do it number of seeds time
for seed_index=1:num_seed
    rand('state', seed(seed_index));

% STEP 1
% generate sub-strings
step = 1;
for i=1:num_substring
```


TABLE 5

Averaged results of three-factor factorial experiment. **Solutions Available** denote the (simulated) solutions that are concretely available in the particular experiment. **Hits (Available)** shows the number of those solutions found. **% Hits (Reference)** shows the fraction of possible solutions in the experiment, out of possible 26 (see Table 2)

Run	[Step 1, type II]	[Step 2, type II]	[Step 3, type II]	Solutions Available	Hits (Available)	%Hits (Reference)
1	0.05	0.05	0.05	17	13	50
2	0.05	0.005	0.05	17	12	46
3	0.05	0.0005	0.05	18	13	50
4	0.05	0.00005	0.05	18	13	50
5	0.05	0.05	0.005	22	16	62
6	0.05	0.005	0.005	21	15	58
7	0.05	0.0005	0.005	21	15	58
8	0.05	0.00005	0.005	21	15	58
9	0.05	0.05	0.0005	22	16	62
10	0.05	0.005	0.0005	21	15	58
11	0.05	0.0005	0.0005	21	15	58
12	0.05	0.00005	0.0005	21	15	58
13	0.05	0.05	0.00005	22	16	62
14	0.05	0.005	0.00005	21	15	58
15	0.05	0.0005	0.00005	21	15	58
16	0.05	0.00005	0.00005	21	15	58
17	0.005	0.05	0.05	19	18	69
18	0.005	0.005	0.05	19	18	69
19	0.005	0.0005	0.05	19	19	73
20	0.005	0.00005	0.05	19	19	73
21	0.005	0.05	0.005	26	25	96
22	0.005	0.005	0.005	26	24	92
23	0.005	0.0005	0.005	26	24	92
24	0.005	0.00005	0.005	26	24	92
25	0.005	0.05	0.0005	26	25	96
26	0.005	0.005	0.0005	26	25	96
27	0.005	0.0005	0.0005	26	25	96
28	0.005	0.00005	0.0005	26	25	96
29	0.005	0.05	0.00005	26	25	96
30	0.005	0.005	0.00005	26	25	96
31	0.005	0.0005	0.00005	26	25	96
32	0.005	0.00005	0.00005	26	25	96

REFERENCES

- [1] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, no. 266, pp. 1021–1024, 1994.
- [2] L. Adleman, P. Rothmund, and et al, "On Applying Molecular Computation to the Data Encryption Standard," *Journal of Computational Biology*, vol. 6, no. 1, pp. 53–63, 1999.
- [3] E. B. Baum and D. Boneh, "Running Dynamic Programming Algorithms on a DNA Computer," in *DNA-Based Computers II: DIMACS*, vol. 44, 1999, pp. 77–87.
- [4] X. Su and L. M. Smith, "Demonstration of a Universal Surface DNA Computer," *Nucleic Acids Research*, vol. 32, no. 10, pp. 3115–3123, 2004.
- [5] Y. Benenson, B. Gil, and et al., "An autonomous Molecular Computer for Logical Control of Gene Expression," *Nature*, vol. 429, no. 6990, pp. 423–429, 2004.
- [6] R. Deaton and M. G. et al, "DNA Computing: A Review," *Fundamenta Informaticae*, vol. 35, no. 1–4, pp. 231–245, 1998.
- [7] K. Langohr, "Sources of Error in DNA Computation," University of Western Ontario, Tech. Rep., 1997.
- [8] G. Gloor, L. Kari, and et al, "Towards a DNA Solution to the Shortest Common Superstring Problem," in *INTSYS '98: Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*. IEEE Computer Society, 1998, p. 140.
- [9] D. Boneh and R. Lipton, "TR-491-95: Making DNA Computers Error Resistant," Princeton University (NJ), Tech. Rep., 1995.
- [10] R. Deaton and R. Murphy, "Good Encodings for DNA-based Solutions to Combinatorial Problems," in *DNA-Based Computers II: DIMACS*, vol. 44, 1999, pp. 247–258.
- [11] J. v. Neuman, "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components," *Annals of Mathematics Studies*, no. 34, 1956.