

Solving Constraint Satisfaction Problems with DNA Computing

Evgeny Dantsin and Alexander Wolpert

School of Computer Science, Roosevelt University
430 South Michigan Ave., Chicago, IL 60605, USA
{edantsin,awolpert}@roosevelt.edu

Abstract. We demonstrate how to solve constraint satisfaction problems (CSPs) with DNA computing. Assuming that DNA operations can be faulty, we estimate error probability of our algorithm. We show that for any k -CSP, there is a polynomial-time DNA algorithm with bounded probability of error. Thus, k -CSPs belong to a DNA analogue of **BPP**.

1 Introduction

After eight years of intensive research in DNA computing it is still not clear whether DNA computing can compete (or will be able to compete in the near future) with existing “silicon” computing. So far problem instances solved with DNA are much smaller than instances of the same problems cracked by electronic computers. Another question that should be addressed is the ability to control errors in DNA computation. Note that electronic computers are faulty too but it is possible to control their errors. In this paper we join the hunt for applications utilizing advantages of DNA computing; in particular we attack the question of error control.

The paper presents a DNA algorithm for solving constraint satisfaction problems (CSPs). Many particular problems that can be stated as CSPs have been already studied in publications on DNA computing. For example, Lipton [10] proposed a DNA algorithm for SAT, Bach et al [4] proposed a DNA algorithm for the 3-colorability problem, etc. We show how to satisfy general constraints with DNA. Our algorithm makes use of the JOIN operation (the name is inspired by “join” in databases, see also [15]). This operation is natural for CSPs as well as for DNA computing. For k -CSPs, the algorithm runs in polynomial time.

The JOIN operation is implemented using well known biochemical DNA manipulations such as EXTRACT, APPEND, MERGE and others, e.g. [5,13,15]. Some of them can introduce errors. For example, EXTRACT can have two kind of errors: false negative error (a strand containing a given substrand is not extracted) and false positive error (a strand not containing a given substrand is extracted). We analyze how such errors affect the result of our algorithm to estimate its probability of error.

To decrease error probabilities, we employ the technique proposed by Karp et al [9]. This method [9] makes EXTRACT error-resilient without a big sacrifice

in the running time. More exactly, EXTRACT is converted into a DNA algorithm whose error probability δ is arbitrary small and whose running time is $O(\log^2 \delta)$. Using this construction, we convert our join-based algorithm into a DNA algorithm with error probability bounded by an arbitrary small constant. For k -CSPs, its running time is still polynomial. That is, any k -CSP can be solved by a polynomial-time DNA algorithm with bounded probability of error. Thus, any k -CSP belongs to a complexity class that can be viewed as a DNA analogue of **BPP** (for **BPP** see e.g. [12]).

The paper is organized as follows. Section 2 contains definitions and notation related to CSPs. In Section 3 we define the JOIN operation and show how to solve CSPs with JOIN. Basic DNA operations used in our model of DNA computation are described in Section 4. Section 5 gives a DNA implementation of our algorithm for CSPs. In Sections 6 and 7 we modify our algorithm into an error-resilient algorithm.

2 Constraint Satisfaction Problems

In a constraint satisfaction problem (CSP) we are given: (i) a finite set of variables that range over a finite domain D , and (ii) constraints C_1, \dots, C_m on values of these variables. We need to determine whether there is a valuation of the variables that satisfies all C_1, \dots, C_m . For example, the satisfiability problem (SAT) can be naturally restated as a CSP in which constraints are disjunctions of literals. Other examples are the graph colorability and solving equations over finite domains.

More formally, each CSP is specified by a finite domain D and a set of predicates defined on D . The predicates are required to be computable in polynomial time. Any atomic proposition $p(x_1, \dots, x_k)$, where p is a predicate symbol and x_1, \dots, x_k are variables ranging over D , is called a *constraint*. A *CSP instance* is a finite set $\{C_1, \dots, C_m\}$ of constraints.

An *assignment* for variables x_1, \dots, x_l is a valuation of these variables in D . We denote such an assignment by $\{x_1 \leftarrow d_1, \dots, x_l \leftarrow d_l\}$ where $d_1, \dots, d_l \in D$. An assignment for one variable is called a *unit assignment*. An assignment for all variables occurring in a CSP instance I is called a *full assignment* for I . A full assignment α for I is called a *solution* to I if α satisfies every constraint in I , i.e., every constraint evaluates to *true* under α . By a *CSP* we mean the following decision problem: given a CSP instance I , determine whether there is a solution to I . Such a problem is called a *k -CSP* if each predicate has its arity at most k , i.e., each constraint contains at most k variables.

Let C be a constraint and α be an assignment for the variables occurring in C . We call α a *solution* to C if α satisfies C . Any constraint C can be identified with the set of all solutions to C .

For an assignment α , the set of all variables occurring in α is denoted by $Var(\alpha)$. For a set S of assignments, we write $Var(S)$ to denote $\cup_{\alpha \in S} Var(\alpha)$.

3 Solving CSPs with Joins

Assignments α_1 and α_2 are called *consistent* if they agree on the common variables in $Var(\alpha_1) \cap Var(\alpha_2)$. We define the *join* of consistent assignments α_1 and α_2 to be the following assignment denoted by $\alpha_1 \bowtie \alpha_2$:

- $\alpha_1 \bowtie \alpha_2$ agrees with α_1 on all variables in $Var(\alpha_1) - Var(\alpha_2)$;
- $\alpha_1 \bowtie \alpha_2$ agrees with α_2 on all variables in $Var(\alpha_2) - Var(\alpha_1)$;
- $\alpha_1 \bowtie \alpha_2$ agrees with both α_1 and α_2 on all variables in $Var(\alpha_1) \cap Var(\alpha_2)$;
- $\alpha_1 \bowtie \alpha_2$ assigns values to only variables in $Var(\alpha_1) \cup Var(\alpha_2)$.

For example, the join of consistent assignments $\{x_1 \leftarrow 0, x_3 \leftarrow 1\}$ and $\{x_2 \leftarrow 1, x_3 \leftarrow 1\}$ is the assignment $\{x_1 \leftarrow 0, x_2 \leftarrow 1, x_3 \leftarrow 1\}$.

We also extend the join operation to sets of assignments. For sets S_1 and S_2 of assignments, we define the *join* operation as follows:

$$S_1 \bowtie S_2 = \{ \alpha \bowtie \beta \mid \alpha \in S_1, \beta \in S_2, \alpha \text{ and } \beta \text{ are consistent} \}$$

In particular, if all pairs α, β are inconsistent, $S_1 \bowtie S_2$ is empty. Our operation is essentially the same as the natural join of relations in databases, e.g. [1]. A similar (but different) DNA operation was introduced in [15].

We present an algorithm that uses the join operation for step-by-step generation of solutions to a CSP instance. Given an instance $\{C_1, \dots, C_m\}$, the algorithm runs in m steps. At step i the algorithm modifies the current set S of assignments (satisfying C_1, \dots, C_{i-1}) in order to satisfy C_i . After m steps the algorithm returns the set S of all solutions to $\{C_1, \dots, C_m\}$.

Algorithm 1 (Join-based algorithm for CSPs).

Input: Sets S_1, \dots, S_m of assignments that represent constraints C_1, \dots, C_m , i.e., each S_i is the set of all solutions to C_i .

Output: The set S of all solutions to the input instance $\{C_1, \dots, C_m\}$.

1. $S \leftarrow S_1$
2. **for** $i \leftarrow 2$ **to** m **do**
 - (a) $S \leftarrow S \bowtie S_i$
 - (b) **if** $S = \emptyset$ **then return** (“no solution”)
3. **return** (S)

The algorithm takes time $O(m \cdot t_{\bowtie})$ where t_{\bowtie} is the maximum running time of the join operation. The space is $O(n \cdot |D|^n)$ where n is the number of variables occurring in $\{C_1, \dots, C_m\}$ and $|D|$ is the number of elements in D . Our next step is to implement Algorithm 1 with DNA.

4 DNA Operations

Most papers on the complexity of DNA algorithms use DNA computation models based on more or less close collections of DNA operations. In this paper, we use a collection of operations similar to the operations in [5].

To solve CSPs with DNA, assignments are encoded by *DNA strands*. For now, a *strand* can be thought of as a string over the alphabet $\Sigma = \{A, C, G, T\}$, see e.g. [2,13]. Given a CSP instance, we fix an encoding of unit assignments by strands (note that the set of all possible unit assignments for the instance is finite). For a unit assignment α , we denote a strand that encodes α by $E(\alpha)$. We assume that all unit assignments are encoded by strands of the same length. Encodings of unit assignments induce encodings of arbitrary assignments: if an assignment α consists of unit assignments $\alpha_1, \dots, \alpha_r$, we encode α by a concatenation of strands $E(\alpha_1), \dots, E(\alpha_r)$.

Strands are contained in *tubes*. We associate with each tube T the set \mathcal{S}_T of all assignments encoded by the strands in T , i.e., \mathcal{S}_T consists of all assignments α such that T contains $E(\alpha)$.

A DNA computation is a computation of a Turing machine augmented by operations on tubes. Operations take tubes and/or assignments (written on Turing machine tapes) as inputs and return tubes and/or assignments.

Extract. This operation takes a tube T and a unit assignment α as input and returns a tube with all strands that contain $E(\alpha)$ as a substrand. The extraction is a variation of the operation SEPARATE which separates the strands in a tube T into two tubes T_1 and T_2 such that T_1 consists of the strands containing a given string $s \in \Sigma^*$ and T_2 contains the rest of T . The extraction operation is denoted by EXTRACT(T, α).

Append. This operation takes a tube T and a unit assignment α as input and returns a tube that contains all strands of T concatenated with $E(\alpha)$. The operation is denoted by APPEND(T, α).

Merge. This operation takes tubes T_1 and T_2 as input and returns a tube that contains all strands of T_1 and T_2 . The operation is denoted by MERGE(T_1, T_2, R).

Detect. This operation is a Boolean operation that returns *true* if an input tube T contains at least one strand and returns *false* otherwise. The operation is denoted by DETECT(T).

Duplicate. This operation takes a tube T as input and returns another tube with the same set of strands as in T without destroying T . The duplication (as well as AMPLIFY) can be implemented with ANNEAL and POLYMERASE, see e.g. [6]. The operation is denoted by DUPLICATE(T).

Create. To start a DNA computation, we need to create the contents of an initial tube. The create operation takes a set S of assignments as input and returns a tube with strands that are encodings of the assignments in S . The operation is denoted by CREATE(S).

5 DNA Implementation of the Join-Based Algorithm

To implement Algorithm 1, we introduce a new DNA operation for computing joins. This operation is denoted by JOIN. We show how to implement JOIN using the basic DNA operations above.

Loosely speaking, the JOIN operation takes a tube T and a set S of assignments as input and returns a tube containing strands that encode $\mathcal{S}_T \bowtie S$. The input also contains additional information, namely the set $Var(\mathcal{S}_T)$ of variables. Of course this set is determined by T , however the computation of \mathcal{S}_T from T with our DNA operations has high complexity. Instead, Algorithm 2 maintains the list $Var(\mathcal{S}_T)$ and passes it to the JOIN operation as part of input.

In the description of Procedure JOIN and Algorithm 2 below, we use the following notation. DNA operations are written using arrows, for example $T_1 \leftarrow \text{EXTRACT}(T, \alpha)$. This means that T_1 contains the result of $\text{EXTRACT}(T, \alpha)$ even if the operation required several tubes and the result was first returned in a different tube. In particular, we write $T \leftarrow \text{EXTRACT}(T, \alpha)$ to denote that the contents of T changes to the result returning by $\text{EXTRACT}(T, \alpha)$. We denote the operation of emptying a tube T by writing $T \leftarrow \emptyset$.

If an assignment α contains a unit assignment for a variable x then this unit assignment is denoted by $\alpha|_x$.

Procedure JOIN(T, S, V)

Input: A tube T with strands; a set S of assignments; the set V of variables occurring in \mathcal{S}_T .

Output: A tube T_1 with strands that encode the join of \mathcal{S}_T and S , i.e., $\mathcal{S}_{T_1} = \mathcal{S}_T \bowtie S$.

1. $T_1 \leftarrow \emptyset$; $U \leftarrow V \cap Var(S)$; $W \leftarrow Var(S) - U$
2. **for each** assignment $\alpha \in S$ **do**
 - (a) $T_2 \leftarrow \text{DUPLICATE}(T)$
 - (b) **for each** variable $x \in U$ **do** $T_2 \leftarrow \text{EXTRACT}(T_2, \alpha|_x)$
 - (c) **if** $\text{DETECT}(T_2)$ **then for each** variable $y \in W$ **do** $T_2 \leftarrow \text{APPEND}(T_2, \alpha|_y)$
 - (d) $T_1 \leftarrow \text{MERGE}(T_1, T_2)$
3. **return** (T_1)

Comment: Step 1 of the procedure divides the set $Var(S)$ of variables into two subsets: the set U of variables "occurring" in T and the set W of variables "not occurring" in T . Then for each assignment α in S , the procedure extracts all strands that "agree" with α on the common variables of U and expands these strands by adding the projection of α onto the "new" variables of W . All extracted and expanded strands are accumulated in a tube T_1 that eventually contains the join of \mathcal{S}_T and S .

Procedure JOIN, we now implement Algorithm 1 with DNA.

Algorithm 2 (DNA implementation of the join-based algorithm).

Input: Sets S_1, \dots, S_m of assignments that represent constraints C_1, \dots, C_m , i.e., each S_i is the set of all solutions to C_i .

Output: A tube T containing strands that encode all solution to $\{C_1, \dots, C_m\}$, i.e., \mathcal{S}_T is the set of all solutions to $\{C_1, \dots, C_m\}$.

1. $T \leftarrow \text{CREATE}(S_1)$; $V \leftarrow \text{Var}(S_1)$
2. **for** $i \leftarrow 2$ **to** m **do**
 - (a) $T \leftarrow \text{JOIN}(T, S_i, V)$; $V \leftarrow V \cup \text{Var}(S_i)$
 - (b) **if not** $\text{DETECT}(T)$ **then return** (“no solution”)
3. **return** (T)

The running time of DNA algorithms is measured by the number of applications of the basic DNA operations. The space complexity is called the *volume* and is measured by the maximum number of strands in all tubes, where the maximum is taken over all steps of the algorithm.

Proposition 1. *Given a k -CSP instance, Algorithm 2 makes $O(mk|D|^k)$ applications of DNA operations, where m is the number of constraints in the instance and $|D|$ is the cardinality of the domain. The volume is at most $|D|^n$ where n is the number of variable in the instance.*

Proof. The algorithm performs one CREATE operation plus $m - 1$ JOIN operations plus $m - 1$ DETECT operations. When running $\text{JOIN}(T, S, V)$, the number of applications of each of DUPLICATE, DETECT, and MERGE is at most $|S|$, which is not greater than $|D|^k$. The number of applications of EXTRACT and APPEND for one assignment $\alpha \in S$ is at most $|\text{Var}(S)|$, which is not greater than k . Therefore, the overall number of applications is $O(mk|D|^k)$. The volume is obviously bounded by the number of all possible assignments, i.e., by $|D|^n$.

6 Probabilistic Nature of DNA Operations

So far we have assumed that DNA computations are *error-free*, i.e., they work perfectly without any errors. However, in reality DNA computations can be faulty because some DNA operations can introduce errors. In particular, among the operations defined in Section 4, the operation APPEND, MERGE, and CREATE are error-free, while EXTRACT, DETECT, and DUPLICATE are faulty.

Extract. It is well known that EXTRACT is probabilistic in nature [3,9,7,11]. Recall that this operation takes a tube T and a unit assignment α as input and returns a tube T_1 with those strands of T that contain the substrand $E(\alpha)$. However, the following errors can occur:

1. A *false negative error*: a strand $s \in T$ containing $E(\alpha)$ does not end up in T_1 . The probability of false negative error is denoted by ϵ and is estimated as $\epsilon \approx 10^{-1}$, see e.g. [11].
2. A *false positive error*: a strand $s \in T$ not containing $E(\alpha)$ ends up in T_1 . The probability of false positive error is denoted by γ and is estimated as $\gamma \approx 10^{-6}$, see e.g. [11].

Single and double strands. To examine DUPLICATE and DETECT, we need to distinguish between different types of strands. By a *single strand* we mean a string over $\Sigma = \{A, C, G, T\}$ together with its *linear orientation*, either with orientation $5' \rightarrow 3'$ or with orientation $3' \rightarrow 5'$, see [14] for details. These two types of strands are denoted by $\uparrow s$ and $\downarrow s$ respectively. A *double strand* $\updownarrow s$ consists of a single strand $\uparrow s$ intertwined with its *Watson-Crick complement* $\downarrow \bar{s}$, see [14]. The operations DUPLICATE and DETECT can be implemented using the following three operations, see [8,6]:

1. Transformation of every pair of single strands $\uparrow s$ and $\downarrow s$ into the double strand $\updownarrow s$.
2. Denaturation of every double strand $\updownarrow s$ into its single strand components $\uparrow s$ and $\downarrow s$.
3. Shortening every strand by a sequence of length l , where l is the length of encodings of unit assignments.

Note that these operations can be regarded as error-free, see e.g. [11,9].

Duplicate. Let T be an input tube for DUPLICATE. Let t be a *tail tag*, i.e., a strand not occurring in encoding of unit assignments. We assume that t has the same length as encodings of unit assignments. To implement DUPLICATE, we append t to every strand in T (using APPEND). Then we apply the transformation of each strand $\uparrow s$ into $\updownarrow s$. Furthermore we apply the denaturation of double strands into their single strand components. Finally, we use EXTRACT to find strands containing the tag t and shorten them to eliminate t . Note that errors of DUPLICATE can appear only because of errors occurring in EXTRACT.

Detect. This operation can be implemented in different ways. In our method, we decide that a tube is not empty if its volume is greater than a threshold volume τ . We double the volume of T until its volume becomes greater than τ . If after $\lceil \log \tau \rceil$ doubles the volume of T is not greater than τ , we decide that T is empty. Since these doubles change the original contents of T , we start DETECT with duplicating of T . Therefore, DETECT can be faulty because of using DUPLICATE.

7 Error-Resilient Computation

Our purpose is to analyze errors of Algorithm 2 and to estimate their probabilities. Like the EXTRACT operation, Algorithm 2 can have two types of error:

1. A *false negative* error: the resulting tube T does not contain an encoding of some solution to the input CSP instance, i.e., a false negative error is the *loss of a solution*.
2. A *false positive* error: the resulting tube T contains an encoding of an assignment that is not a solution to the input CSP instance, i.e., a false positive error is the *acquisition of a pseudo-solution*.

7.1 Probabilities of False Negative Error and False Positive Error

Clearly, Algorithm 2 returns its result with a false negative error if at least one of the operations EXTRACT, DUPLICATE, or DETECT introduces a false negative error. Recall that the algorithm invokes $\text{JOIN}(T, S_i, V)$ for $i = 2, \dots, m$ and also invokes $\text{DETECT}(T)$. In turn, each run of $\text{JOIN}(T, S_i, V)$ includes:

- $|S_i|$ runs of DUPLICATE; let \mathcal{E}_{dup}^i be the event that at least one of these runs has a false negative error;
- $|S_i|$ runs of DETECT; let \mathcal{E}_{det}^i be the event that at least one of these runs has a false negative error;
- at most $|S_i| \cdot k$ runs of EXTRACT; let \mathcal{E}_{ext}^i be the event that at least one of these runs has a false negative error.

Now we estimate probabilities $\Pr[\mathcal{E}_{dup}^i]$, $\Pr[\mathcal{E}_{det}^i]$, and $\Pr[\mathcal{E}_{ext}^i]$. Suppose that for each run of EXTRACT and for each strand s , the probability that s contains a given substrand but does not end up in the resulting tube is not greater than ϵ . Then we have

$$\begin{aligned}\Pr[\mathcal{E}_{dup}^i] &\leq 2\epsilon |D|^n \\ \Pr[\mathcal{E}_{det}^i] &\leq 2\epsilon |D|^n \\ \Pr[\mathcal{E}_{ext}^i] &\leq k\epsilon |D|^n\end{aligned}$$

The first inequality holds because DUPLICATE involves two EXTRACTS, and any test tube contains at most $|D|^n$ strands. The second inequality holds because DETECT invokes one DUPLICATE. The third inequality holds because \mathcal{E}_{ext}^i happens if at least one of k consecutive EXTRACTS introduces a false negative error. Note that the number of strands in any tube does not exceed $|D|^n$. The probability that a false negative error occurs in $\text{JOIN}(T, S_i, V)$ is not greater than

$$\Pr[\mathcal{E}_{dup}^i \cup \mathcal{E}_{det}^i \cup \mathcal{E}_{ext}^i] \leq (4 + k)\epsilon |D|^n$$

Finally, the probability that Algorithm 2 has a false negative error is the sum of the probabilities of false negative error for all $\text{JOIN}(T, S_i, V)$ where $i = 2, \dots, m$ plus the probability of false negative error of the final DETECT. Thus we have for Algorithm 2:

$$\Pr[\text{Algorithm 2 has a false negative error}] \leq ((m - 1)(4 + k) + 2)\epsilon |D|^n \quad (1)$$

The probability that Algorithm 2 has a false positive error can be estimated in a similar way. Assume that for each run of EXTRACT and for each strand s , the probability that s does not contain a given substrand but ends up in the resulting tube is not greater than γ . Then, repeating the arguments above, we have

$$\Pr[\text{Algorithm 2 has a positive error}] \leq (((m - 1)(4 + k) + 2)\gamma |D|^n) \quad (2)$$

These bounds $O(mk\epsilon|D|^n)$ and $O(mk\gamma|D|^n)$ use the upper bound $|D|^n$ on the number of strands in any tube. Instead, we could estimate the number of strands at each step i : when $\text{JOIN}(T, S_i, V)$ is performed, the number of strands in a tube does not exceed $|D|^{ik}$. Then we would have the bounds $O(k\epsilon|D|^{mk})$ and $O(k\gamma|D|^{mk})$ respectively.

7.2 Bounded Probabilities of Error

The above bounds on probabilities of error depend on ϵ and γ . Is it possible to modify EXTRACT so that these probabilities decrease? The answer is yes. It is shown in [9] that for any given constant δ , the operation EXTRACT can be converted into a DNA algorithm such that its false negative and false positive errors are not greater than δ . The running time of this algorithm depends on ϵ , γ , and δ , namely the complexity of the algorithm is $\Theta(\lceil \log_\epsilon \delta \rceil \lceil \log_\gamma \delta \rceil)$. This result is proved in [9] for a slightly less general setting: strands are encodings of bit strings, and EXTRACT returns strands that encode strings containing a given bit in a given position. However, this case can be easily generalized for our settings.

Proposition 2 (Karp et al [9]). *There exists a DNA algorithm ER-EXTRACT (error-resilient extraction) with the following properties:*

1. *The algorithm runs on the following input: a tube T , a unit assignment α , and a number δ such that $0 < \delta < 1$. The algorithm returns a tube T_1 .*
2. *For each strand $s \in T$ such that s contains $E(\alpha)$, the probability that s does not end up in T_1 is not greater than δ .*
3. *For each strand $s \in T$ such that s does not contain $E(\alpha)$, the probability that s ends up in T_1 is not greater than δ .*
4. *Assuming that the basic DNA operations EXTRACT and MERGE run in constant time, the algorithm ER-EXTRACT runs in time $\Theta(\lceil \log_\epsilon \delta \rceil \lceil \log_\gamma \delta \rceil)$, where ϵ and γ have the same meaning as above.*

Proof. Straightforward generalization of the proof of [9, Theorem 3.1].

Using Proposition 2, we can solve a k -CSP in polynomial time with bounded two-sided error probability. More exactly, the probabilities of false negative and false positive errors are both less than a constant $c < 1/2$. The constant c does not depend on the input size. Loosely speaking, we show that any k -CSP belongs to a DNA counterpart of **BPP** (for **BPP** see e.g. [12]).

Proposition 3. *For any k -CSP \mathcal{P} , there is a DNA algorithm $\mathcal{A}_{\mathcal{P}}$ that solves \mathcal{P} in polynomial time with bounded error probability. Namely, the probabilities of false negative and false positive errors of $\mathcal{A}_{\mathcal{P}}$ are less than $1/4$.*

Proof. We obtain $\mathcal{A}_{\mathcal{P}}$ from Algorithm 2 by replacing every EXTRACT operation (including those in DUPLICATE and DETECT) by the ER-EXTRACT algorithm. Namely, we take ER-EXTRACT with

$$\delta \leq \frac{1}{4((m-1)(4+k)+2)|D|^n}. \quad (3)$$

Then, according to inequalities (1) and (2), we have

$$\begin{aligned} \Pr[\mathcal{A}_{\mathcal{P}} \text{ has a false negative error}] &\leq ((m-1)(4+k)+2)\delta|D|^n \leq 1/4 \\ \Pr[\mathcal{A}_{\mathcal{P}} \text{ has a false positive error}] &\leq ((m-1)(4+k)+2)\delta|D|^n \leq 1/4 \end{aligned}$$

It remains to estimate the running time of $\mathcal{A}_{\mathcal{P}}$. It follows from Proposition 2 that ER-EXTRACT with δ satisfying (3) runs in time

$$\begin{aligned}\Theta(\lceil \log_{\epsilon} \delta \rceil \lceil \log_{\gamma} \delta \rceil) &= \Theta(\lceil \log_{\epsilon}(m^{-1}|D|^{-n}) \rceil \lceil \log_{\gamma}(m^{-1}|D|^{-n}) \rceil) \\ &= \Theta(n^2 \log^2 m).\end{aligned}$$

Since Algorithm 2 uses a polynomial number of applications of EXTRACT (Proposition 1), we have a polynomial bound on the running time of $\mathcal{A}_{\mathcal{P}}$.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 1994.
3. L. M. Adleman. On constructing a molecular computer. In R. Lipton and E. Baum, editors, *DNA Based Computers*, volume 27 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1995.
4. E. Bach, A. Condon, E. Glaser, and C. Tanguay. DNA models and algorithms for NP-complete problems. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, pages 290–300, 1996.
5. R. Beigel and B. Fu. Solving intractable problems with DNA computing. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity*, pages 154–169, 1998.
6. D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. In R. Lipton and E. Baum, editors, *DNA Based Computers*, volume 27 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 37–66. American Mathematical Society, 1995.
7. D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. Making DNA computers error resistant. In *DNA Based Computers II*, volume 44 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–170. American Mathematical Society, 1996.
8. K. Chen and V. Ramachandran. A space-efficient randomized DNA algorithm for k -SAT. In *Proceedings of the 6th International Workshop on DNA-Based Computers*, pages 199–208, 2000.
9. R. M. Karp, C. Kenyon, and O. Waarts. Error-resilient DNA computation. *Random Structures and Algorithms*, 15(3-4):450–466, 1999.
10. R. J. Lipton. DNA solutions of hard combinatorial problems. *Science*, 268:542–548, April 1995.
11. C. C. Maley. DNA computation: theory, practice, and prospects. *Evolutionary Computation*, 6(3):201–230, 1998.
12. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
13. G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer, 1998.
14. P. A. Pevzner. *Computational Molecular Biology*. MIT Press, 2000.
15. J. H. Reif. Parallel molecular computation: models and simulations. *Algorithmica*, 25(2):142–176, 1999.