Molecular Computing Solutions of some Classical Problems

István Katsányi*

Abstract

In this paper we give efficient molecular computing solutions to seven well-known NP-complete problems, namely the Hamiltonian circuit, Path with forbidden pairs, Longest path, Monochromatic triangle, Partition into triangles, Partition into paths of length two, Circuit satisfiability problems in the Parallel filtering model of Amos et al.

Key words: Molecular computing, DNA computing, NP-complete problems, Parallel filtering model

1 Introduction

In the last decade molecular biology has become one of the fastest growing discipline in the world. Some of the results are widely known, let us only mention the major breakthroughs in the Human Genome Project and nanotechnology. The progress made possible the birth of a new branch of science, that is called molecular computing (or DNA computing). Leonard M. Adleman published a paper [1] in 1994, which later become the foundation-stone of this new subject. In his article Adleman demonstrates how can one solve the classical NP-complete Hamiltonian Path Problem in polynomial time using DNA strands and the techniques of molecular biology. One month later Richard J. Lipton pointed out in [2], that although theoretically it would be enough for showing the superiority of the molecular computing paradigm to solve a single NP-complete problem in polynomial time, it is important to give direct solutions to more problems in NP, because the reduction of one problem to the Hamiltonian Path Problem could be non-trivial and time consuming. Therefore he gave his solution to the very important *Boolean Satisfiability Problem (SAT)*. Since then many classical problems were solved in one of the emerged theoretical models of molecular computing. Some of these solutions are summarized in Table 1.

In this technical report algorithms for NP-complete problems are described, that (with the exception of the Circuit satisfiability problem) were not investigated before. All of these algorithms need polynomial time for processing their input even in the worst case. We use the so-called *Parallel filtering model*, originally published by Amos, Gibbons and Hodgson in [7]. This model is very simple and all its operations refer to feasible laboratory operations.

*Eötvös Loránd University, Department of Algorithms and Applications, 1117 Budapest, Pázmány Péter sétány 1/C, e-mail: kacsa@ludens.elte.hu

Problem	when solved	reference
Hamiltonian path problem	1994	[1]
Boolean satisfiability	1994	[2]
3-colouring	1995	[3]
Quantified Boolean formulae	1995	[4]
Independent set	1996	[5]
Knapsack	1996	[6]
Subgraph isomorphism	1996	[7]
Maximum clique	1996	[7]
MAX-CNF satisfiability	1996	[8]
Circuit satisfiability	1996	[8],[9]
$(3-2)$-system	1997	[10]
Shortest common superstring	1998	[11]
Bounded Post correspondence	2000	[12]

Table 1: Some problems and their first molecular computing solutions

2 Preliminaries

When not stated otherwise, we will use the standard notations used in the theory of formal languages (see for example [13]). The length of a word u will be denoted by $|u|$. The notation of the empty word is λ. We will use the usual big-oh ($\mathcal{O}$) notation for an asymptotic upper bound of a complexity function.

For a graph $G = (V, E)$ with vertex set V and edge set E, a *simple path* in G is a sequence $v_1, v_2, \ldots, v_k$ of distinct vertices from V such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i < k$. A *simple circuit* in G is a sequence $v_1, v_2, \ldots, v_k$ of distinct vertices from V such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i < k$ and $\{v_k, v_1\} \in E$. A *Hamiltonian path* in G is a simple path that includes all the vertices of G. A *Hamiltonian circuit* in G is a simple circuit that includes all the vertices of G.

Now we briefly define the *Parallel filtering model* we use in the algorithms. The model was first published in [7], see also [14] and [15]. In this model, a computation works with finite multisets of strings over a given alphabet. The first operation of any computation is the creation of an *initial multiset*. There are no formal restrictions for the initial multiset, but it must be "easy to generate" by terms of laboratory operations. Typically, the initial multiset consists of strings of equal length, which is a linear function of the input size of the problem to be solved. This multiset contains all possible solutions of the given problem, encoded in strings. After inputting the initial multiset, the computation proceeds with performing certain operations on the multisets generated. These operations are intended to filter out those strings, which cannot be a solution. As a result of the executed operations several multisets may exist at the same time. The result of a computation is either a single multiset, an answer like *yes/no/empty*, or the string value of a random member of a generated multiset.

The following operations are allowed in the computation:

- $Input(T)$ creates the initial multiset.
- $Remove(T, S)$ removes from the multiset T any string that contains at least one occurrence of any string of the non-empty set of strings S.

- $Union(\{T_1, T_2, \ldots, T_n\}, T)$. The set T is created, which is the multiset union of the distinct multisets $T_1, T_2, \ldots, T_n$, where n is an arbitrary non-negative integer.
- $Copy(T, \{T_1, T_2, \ldots, T_n\})$. This operation creates n duplicates of the multiset T and places it in the multisets $T_1, T_2, \ldots, T_n$. (n is an arbitrary positive integer.)
- $Select(T)$ selects and returns an element of T at random. If T is the empty set then *empty* is returned. This operation is only executed at the end of a computation.

Each of these operations can be effectively implemented on DNA strands. Originally it was assumed that the evaluation of the operations take constant time, but in [14] the authors introduced the so-called *strong model*, where the time complexity of the *Remove*, *Union* and *Copy* operations is dependent on the actual parameters. The time complexity of the $Remove(T, S)$ operation is proportional to the cardinality of the set S of the substrings to be removed. The $Copy(T, \{T_1, T_2, \ldots, T_n\})$ and the $Union(\{T_1, T_2, \ldots, T_n\}, T)$ operations have time complexity proportional to the integer value n. For complexity investigations we will use this more realistic strong model.

We will denote the operation, which assigns the empty value to a set T by $T := \emptyset$ instead of $Union(\emptyset, T)$.

3 Results

In the algorithms given in this article, the alphabet

$$\Sigma_n^k = \{p_1, p_2, \ldots, p_k, a_1, a_2, \ldots, a_n\}$$

and the initial set $T_n^k = \{p_1 a_{i_1} p_2 a_{i_2} \ldots p_k a_{i_k} | 1 \leq i_j \leq n \ (j = 1, 2, \ldots, k)\}$ will be used for some positive integers n and k, where $k \leq n$. The indexed p symbols denote the *positions* within a word and the indexed a symbols denote an integer value between 1 and n. It will be important that the integer values are separated by position markers.

In [7] it is shown, that the set $P_n \subset T_n^n$, which is an encoding of all permutations of the integers $\{1, 2, \ldots, n\}$ can be generated efficiently in the Parallel filtering model if T_n^n is given. Formally, the set

$$P_n = \{p_1 a_{i_1} p_2 a_{i_2} \ldots p_n a_{i_n} \in T_n^n | \text{for all } 1 \leq j, j' \leq n \ : \ i_j \neq i_{j'} \text{ whenever } j \neq j'\}$$

can be obtained from T_n^n by using the operations of the model in $\mathcal{O}(n^2)$ time. Hence, in addition to T_n^k, we will also use P_n as initial set for our algorithms.

We quote here an example algorithm of [7]:
Problem name: Hamiltonian path
Input: An n-node undirected graph $G = (V, E)$.
Question: Is there a simple path in G that contains every node in V?
Let $V = \{v_1, \ldots, v_n\}$. The node-sequences of G are represented by strings in P_n. The string $p_1 a_{i_1} p_2 a_{i_2} \ldots p_n a_{i_n}$ represents the node-sequence $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$. Such a node sequence is a Hamiltonian path iff it is a path. That is, the consecutive nodes in the sequence must be connected by an edge in E.

HAMILTONIAN-PATH
1 $Input(T = P_n)$
2 **for** $2 \leq i \leq n$ and $1 \leq j, j' \leq n$ such that $\{j, j'\} \notin E$ **in parallel do**
3 $Remove(T, \{a_j p_i a_{j'}\})$
4 **end**
5 $Select(T)$

The time complexity of the algorithm is $\mathcal{O}(n^3)$, including the creation of P_n. Please note that the *Remove* operations can be formulated as a single operation:

$$Remove(T, \{a_j p_i a_{j'} \mid 2 \leq i \leq n,\ 1 \leq j, j' \leq n \text{ and } \{j, j'\} \notin E\}).$$

We are now ready to state the main theorem of this paper.

Theorem 1 *The following problems can be efficiently solved in the Parallel filtering model: Hamiltonian circuit, Longest path, Path with forbidden pairs, Monochromatic triangle, Partition into triangles, Partition into paths of length two, Circuit satisfiability.*

The definitions of the problems and the proofs can be found in the subsequent subsections. The proof is always a construction of an algorithm in the Parallel filtering model which solves the given problem. More detailed descriptions of the problems can be found in [16] and in [17].

Hamiltonian circuit

Input: An n-node undirected graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$.
Question: Does G contain a Hamiltonian circuit?

For an integer i between 1 and n, let us denote by U_i the node-sequences $s_1, s_2, \ldots, s_n$ which satisfy the following conditions:

(i) for each pair $j, j' \in [1, n]$, $j \neq j'$ implies $s_j \neq s_{j'}$,

(ii) for each $j = 1, 2, \ldots, n-1$ the pair $\{s_j, s_{j+1}\}$ is in E,

(iii) $s_1 = v_i$, and

(iv) the pair $\{s_n, v_i\}$ is in E.

Let U be the union of the sets U_i $(i = 1, 2, \ldots, n)$. Clearly, G possesses a Hamiltonian circuit exactly in the case when U is not empty.

The algorithm uses the same representation as the one in the example problem HAMILTONIAN-PATH. Hence the set T after step 1 of the algorithm contains only strings that encode node sequences, which satisfy condition (i). As the algorithm progresses, more and more conditions of the above four will be satisfied.

Step 2 removes all strings from the set T, which encode such node sequences, which do not satisfy condition (ii). The time complexity of this step is $\mathcal{O}(n^3)$. In step 3 we make n copies of T in order to produce later the sets T_i, which

HAMILTONIAN-CIRCUIT

1 $Input(T = P_n)$
2 $Remove(T, \{a_j p_i a_{j'} \mid 2 \leq i \leq n,\ 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
3 $Copy(T, \{T_1, \ldots, T_n\})$
4 **for each** $i = 1, 2, \ldots, n$ **in parallel do**
5 $\quad Remove(T_i, \{p_1 a_j \mid 1 \leq j \leq n \text{ and } j \neq i\})$
6 $\quad Remove(T_i, \{p_n a_j \mid 1 \leq j \leq n \text{ and } \{v_j, v_i\} \notin E\})$
7 **end**
8 $Union(\{T_1, \ldots, T_n\}, T)$
9 $Select(T)$

encode exactly the node sequences in U_i $(i = 1, 2, \ldots, n)$. We perform the *Remove* operations in step 5 and step 6 parallel for each $i = 1, 2, \ldots, n$. (In fact, we can combine step 5 and step 6 into a single *Remove* operation.) After step 5 and step 6, the strings in each T_i $(i = 1, 2, \ldots, n)$ encodes sequences, which fulfil conditions (i)–(iii) and (i)–(iv), respectively. The time complexities of both step 5 and step 6 are $\mathcal{O}(n)$. After step 8 of the algorithm the strings in T encodes the node sequences in U, hence the terminating 9th step gives the correct answer: an encoding of a possible Hamiltonian circuit, if there exists one, or else the negative answer *empty*. The total time complexity of the algorithm is determined by step 2 of the algorithm, hence we get an overall complexity of $\mathcal{O}(n^3)$.

Note: The algorithm is the same for directed graphs. This holds true for all of the algorithms given in this article.

Longest path

Input: An n-node undirected graph $G = (V, E)$, two distinct nodes s and t in V, a positive integer $k \leq n - 1$.
Question: Is there a simple path between s and t in G that contains at least k edges?

The representation is the same as the one used in the algorithm HAMILTONIAN-PATH, the algorithm is also similar, but at a time we only deal with the first few nodes of the node-sequence represented by the strings in P_n. The subsequence is checked if it is a valid path in G, and if it begins and ends in the given nodes. We may assume, that $V = \{v_1, \ldots, v_n\}$, where $s = v_1$ and $t = v_n$.

For an integer i between k and n, let us denote by U_i the node-sequences $s_1, s_2, \ldots, s_n$ of length n which satisfy the following conditions:

(i) for each pair $j, j' \in [1, n]$, $j \neq j'$ implies $s_j \neq s_{j'}$,

(ii) $s_1 = v_1$, and

(iii) for each $j = 1, 2, \ldots, i - 1$ the pair $\{s_j, s_{j+1}\}$ is in E,

Let us denote by U_i' those node-sequences $s_1, s_2, \ldots, s_i, \ldots, s_n$ of U_i, where $s_i = v_n$. These sequences encode simple paths of the graph G between nodes $s = v_1$ and $t = v_n$ that contains exactly $i - 1$ edges. Let U be the union of

the sets $U'_{k+1}, U'_{k+2}, \ldots, U'_n$. Clearly, the problem has a solution exactly in the case when U is not empty. In the following algorithm the sets encoding the node-sequences of U'_i $(i = k+1, k+2, \ldots, n)$ are iteratively created, and the set T_r always contains the coding of $\cup_{i=k+1}^{k'} U'_i$ for the actual value of the loop variable k'.

LONGEST-PATH

1 $Input(T = P_n)$
2 $Remove(T, \{p_1 a_j \mid 2 \leq j \leq n\})$
3 $Remove(T, \{a_j p_i a_{j'} \mid 2 \leq i \leq k,\ 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
4 $T_r := \emptyset$
5 **for** $k' \leftarrow k+1$ **to** n **do**
6 $Remove(T, \{a_j p_{k'} a_{j'} \mid 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
7 $Copy(T, \{T'\})$
8 $Remove(T', \{p_{k'} a_j \mid 1 \leq j \leq n-1\})$
9 $Union(\{T_r, T'\}, T_r)$
10 **end**
11 $Select(T_r)$

Steps 1–4 initialize the sets T and T_r, steps 5–10 form a loop, step 11 selects a possible solution, if there is one. In the beginning of each cycle, T encodes $U_{k'-1}$, and T_r encodes $\cup_{i=k+1}^{k'-1} U'_i$. By the end of a cycle the former condition remains true for a k' greater by one to its original value.

Step 1 ensures, that the strings in T encode sequences that satisfy condition (i). For the sake of simplicity, we will say, that a string satisfies some condition, if the string encodes a sequence, which satisfies the given condition. After step 2, only strings satisfying condition (ii) remain in T. After step 3, strings in T encodes exactly the sequences in U_k, hence step 4 ensures that the above loop invariant is true at the beginning of the first cycle.

After step 6, strings in T encodes exactly the sequences in $U_{k'}$. We create a copy of it in T' in step 7, and remove those strings in step 8, which do not belongs to $U'_{k'}$. We complete the set T_r with this set in step 9, hence the loop invariant remains true for the incremented value of k'. By the time we reach step 10 of the algorithm, T_r encodes the set U, so the $Select(T_r)$ operation gives a possible solution of the given problem, or answers *empty*, which means that there is no solution.

It is easy to see that the time complexity of step 3 is $\mathcal{O}((k-1) * n^2)$. One execution of step 6 takes $\mathcal{O}(n^2)$ time, the loop performs $n-k$ cycle, so altogether step 6 takes $\mathcal{O}((n-k)n^2)$ time. The other steps need less time, so the overall complexity is $\mathcal{O}(n^3)$.

Note 1 *If we change step* 1 *to* $Input(T = T_n^n)$*, then we get the solution to a variation of the Longest path problem, where the term "simple path" is changed to "path" in the definition.*

Longest circuit

Input: An n-node undirected graph $G = (V, E)$ and a positive integer $k \leq n$.
Question: Does G contain a simple cycle containing at least k edges?

Since the algorithm is a simple combination of the former two programs, we omit the details of the verification. The time complexity is still $\mathcal{O}(n^3)$.

LONGEST-CIRCUIT

1 $Input(T = P_n)$
2 $Remove(T, \{a_j p_i a_{j'} \mid 2 \leq i \leq k-1,\ 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
3 $T_r := \emptyset$
4 **for** $k' \leftarrow k$ **to** n **do**
5 $\quad Remove(T, \{a_j p_{k'} a_{j'} \mid 1 \leq j, j' \leq n \text{ and } \{v_j, v_{j'}\} \notin E\})$
6 $\quad Copy(T, \{T_1, \ldots, T_n\})$
7 $\quad$ **for each** $i = 1, 2, \ldots, n$ **in parallel do**
8 $\quad\quad Remove(T_i, \{p_1 a_j \mid 1 \leq j \leq n \text{ and } j \neq i\})$
9 $\quad\quad Remove(T_i, \{p_{k'} j \mid 1 \leq j \leq n \text{ and } \{v_j, v_i\} \notin E\})$
10 $\quad$ **end**
11 $\quad Union(\{T_r, T_1, T_2, \ldots, T_n\}, T_r)$
12 **end**
13 $Select(T_r)$

Path with forbidden pairs

Input: An n-node directed graph $G = (V, A)$, two distinct nodes s and t belonging to V and a finite collection C of r pairs of distinct nodes from V.
Question: Is there a directed path from node s to node t in G that contains at most one node from each pair of nodes in the collection C?

The PATH-WITH-FORBIDDEN-PAIRS algorithm is a modification of procedure LONGEST-PATH, the used coding is exactly the same.

Let $V = \{v_1, v_2, \ldots, v_n\}$. We may assume, that $s = v_1$ and $t = v_n$. Obviously the length of a path that satisfies the conditions of the problem is at most $n - r$, hence we use as input the set $T = T_n^{n-r}$, which encodes the sequences of length $n-r$ of the integers $1, \ldots, n$. It can be immediately seen, that the differences of this problem and the Longest path problem are the following: (i) we are searching for paths, instead of simple paths, (ii) the constant k of the Longest path problem has the value 1, and (iii) there is an additional condition for the solution paths. By *Note 1* it is enough to show that this additional condition can be checked efficiently in the Parallel Filtering model, which is obvious, if we observe, that the paths that contain at most one of a given (v_i, v_j) pair of nodes is the union of the paths that do not contain the node v_i and those that do not contain the node v_j. We only have to check all paths for all pairs in C one after the other and we get a procedure to filter out the unwanted paths. This property has to be checked for only those part of the examined node-sequence, which take part in the path from node s to node t. However, we may check this

condition for the whole node-sequence, because we can choose the nodes after the first occurrence of the node t (if it exists in the sequence) in such a way that does not violate the checked condition. (For example these nodes can be set to t.) To be more precise, for all node-sequences $s_1, s_2, \ldots, s_k$ with $k < n - r$, that encode a solution to the problem there exist nodes $s_{k+1}, \ldots, s_{n-r}$ such that $s_1, s_2, \ldots, s_{n-r}$ contains at most one node from each pair of nodes in the collection C.

In the algorithm PATH-WITH-FORBIDDEN-PAIRS steps 2–7 perform the above filtering procedure, steps 8–15 search for a path of length at least 1 as in the algorithm LONGEST-PATH. Step 1 provides the original input, and step 16 takes care of the final read-out of the solution.

PATH-WITH-FORBIDDEN-PAIRS

1 $Input(T = T_n^{n-r})$
2 **for** all pairs $(v_i, v_j) \in C$ **do**
3 $Copy(T, \{T_1, T_2\})$
4 $Remove(T_1, \{a_i\})$
5 $Remove(T_2, \{a_j\})$
6 $Union(\{T_1, T_2\}, T)$
7 **end**
8 $Remove(T, \{p_1 a_j \mid 2 \leq j \leq n\})$
9 $T_r := \emptyset$
10 **for** $k' \leftarrow 2$ **to** $n - r$ **do**
11 $Remove(T, \{a_j p_{k'} a_{j'} \mid 1 \leq j, j' \leq n \text{ and } (v_j, v_{j'}) \notin A\})$
12 $Copy(T, \{T'\})$
13 $Remove(T', \{p_{k'} a_j \mid 1 \leq j \leq n - 1\})$
14 $Union(\{T_r, T'\}, T_r)$
15 **end**
16 $Select(T_r)$.

Since steps 2–7 only need $\mathcal{O}(r)$ time, the time complexity is dominated by the time complexity of the algorithm LONGEST-PATH, which is $\mathcal{O}(n^3)$.

Monochromatic triangle

Input: An undirected graph $G = (V, E)$.
Question: Can the edges of G be partitioned into two disjoint sets E_1 and E_2, in such a way that neither of the two graphs $G_1 = (V, E_1)$ or $G_2 = (V, E_2)$ contains a triangle, i.e. a set of three distinct nodes u, v, w such that $\{u, v\}$, $\{u, w\}$, $\{v, w\}$ are all edges?

Let us assume, that we have n edges and $E = \{e_1, e_2, \ldots, e_n\}$. We will use the initial set T_2^n. The string $s = p_1 a_{i_1} p_2 a_{i_2} \ldots p_n a_{i_n} \in T_2^n$ represents such a partition, where an arbitrary edge $e_j \in E_1$ if $i_j = 1$, that is, the string $p_j a_1$ is a substring of s, otherwise $e_j \in E_2$. Clearly, T_2^n represents all the possible partitions, we just have to filter out those strings, which do not satisfy the conditions of the problem.

The final solution will consist of strings where for each triangle of G made of the edges e_i, e_j and e_k $(i < j < k)$ either

(i) $e_i \in E_1$ and $e_j \in E_2$, or

(ii) $e_i \in E_2$ and $e_j \in E_1$, or

(iii) both $e_i, e_j \in E_1$ but $e_k \in E_2$, or

(iv) both $e_i, e_j \in E_2$ but $e_k \in E_1$.

The MONOCHROMATIC-TRIANGLE algorithm creates the four sets that satisfies the above four conditions for a triangle of G, then proceeds to the next triangle considering only the union of the mentioned four sets.

MONOCHROMATIC-TRIANGLE

1 $Input(T = T_2^n)$
2 **for each** $1 \leq i < j < k \leq n$, such that
edges e_i, e_j and e_k form a triangle in G **do**
3 $Copy(T, \{T_b, T_w\})$
4 $Remove(T_b, \{p_i a_1\})$
5 $Remove(T_w, \{p_i a_2\})$
6 $Copy(T_b, \{T_{bb}, T_{bw}\})$
7 $Remove(T_{bb}, \{p_j a_1\})$
8 $Remove(T_{bw}, \{p_j a_2\})$
9 $Copy(T_w, \{T_{wb}, T_{ww}\})$
10 $Remove(T_{wb}, \{p_j a_1\})$
11 $Remove(T_{ww}, \{p_j a_2\})$
12 $Copy(T_{bb}, \{T_{bbw}\})$
13 $Remove(T_{bbw}, \{p_k a_2\})$
14 $Copy(T_{ww}, \{T_{wwb}\})$
15 $Remove(T_{wwb}, \{p_k a_1\})$
16 $Union(\{T_{wb}, T_{bw}, T_{wwb}, T_{bbw}\}, T)$
17 **end**
18 Select(T)

For the sake of simplicity let us call the nodes in E_1 *white*, and the nodes in E_2 *black*. Steps 3–5 create the sets T_b and T_w that contains those strings of T, which encode such colourings, where the ith node is black or white, respectively. (Note that a node is black, if it is not white and vice versa.) In steps 6–8 the sets T_{bb} and T_{bw} are created, which encode those sequences, where the ith node is black, and the jth node is black or white, respectively. Note that the set T_{bw} contains exactly those strings of T, which satisfy the above (ii) condition. The set T_{bb} is a superset of those strings of T, which satisfies condition (iv). In steps 9–11 we get the sets T_{wb} and T_{ww}, which are in the same situation with respect to conditions (i) and (iii). In steps 12–15 the sets T_{bbw} and T_{wwb} are created, which encode exactly those strings of T, which satisfies condition (iv) and (iii), respectively. In step 16 the set T gets a value which is a subset of its original value: it encode colourings, where the triangle $\{e_i, e_j, e_k\}$ is not monochromatic.

If we do this filtering for all triangles of G and T is still not the empty set, then step 18 selects a possible colouring, otherwise it answers *empty*.

Since all used operations have time complexity $\mathcal{O}(1)$, the time complexity of the algorithm is proportional to the number of triangles in G. Please note that the sets T_b and T_w computed in steps 3–5 are independent of the value of the integers j and k, hence with a slight modification of the algorithm, it can be achieved to compute them only once for each value of i. However, the asymptotic time complexity of the algorithm is not altered by this change.

Partition into triangles

Input: A graph $G = (V, E)$, with $|V| = 3q$ for a positive integer q.
Question: Is there a partition of V into q disjoint sets $V_1, \ldots, V_q$ of three vertices each such that, for each $V_i = \{u_i, v_i, w_i\}$ the three edges $\{u_i, v_i\}$, $\{v_i, w_i\}$ and $\{w_i, u_i\}$ all belong to E?

The partitions are represented by strings in P_{3q}. The string

$$p_1 a_{i_1} p_2 a_{i_2} \ldots p_{3q} a_{i_{3q}}$$

represents the partition where $V_1 = \{v_{i_1}, v_{i_2}, v_{i_3}\}$, $V_2 = \{v_{i_4}, v_{i_5}, v_{i_6}\}$, $\ldots$, $V_q = \{v_{i_{3q-2}}, v_{i_{3q-1}}, v_{i_{3q}}\}$. We can filter out the unwanted strings by a single (but rather complex) *Remove* operation.

PARTITION-INTO-TRIANGLES

1 $Input(T = P_{3q})$
2 $Remove(T, \{p_{3i-2}a_j p_{3i-1}a_{j'} p_{3i}a_{j''} \mid 1 \leq i \leq q, \quad 1 \leq j, j', j'' \leq 3q$
and at least one of the edges $\{v_j, v_{j'}\}, \{v_{j'}, v_{j''}\}$ and $\{v_{j''}, v_j\}$
does not belong to $E\}$)
3 Select(T)

The time complexity of the PARTITION-INTO-TRIANGLES algorithm is proportional to the strings in the second operand of the *Remove* operation, which is $\mathcal{O}(q^4)$.

Partition into paths of length two

Input: A graph $G = (V, E)$, with $|V| = 3q$ for a positive integer q.
Question: Is there a partition of V into q disjoint sets $V_1, \ldots, V_q$ of three vertices each such that, for each $V_i = \{u_i, v_i, w_i\}$ at least two of the three edges $\{u_i, v_i\}$, $\{v_i, w_i\}$ and $\{w_i, u_i\}$ belong to E?

The PARTITION-INTO-PATHS-OF-LENGTH-TWO algorithm uses the same representation as the one in PARTITION-INTO-TRIANGLES. The procedure is also similar, the only difference is that the filtering condition is changed in compliance with the altered condition. The time complexity is $\mathcal{O}(q^4)$ again.

PARTITION-INTO-PATHS-OF-LENGTH-TWO

1 $Input(T = P_{3q})$
2 $Remove(T, \{p_{3i-2}a_j p_{3i-1}a_{j'}p_{3i}a_{j''} \mid 1 \leq i \leq q, \quad 1 \leq j, j', j'' \leq 3q$
and at least two of the edges $\{v_j, v_{j'}\}, \{v_{j'}, v_{j''}\}$ and $\{v_{j''}, v_j\}$
does not belong to $E\}$)
3 Select(T)

Circuit satisfiability

Input: A directed acyclic graph $G = (V, E)$ representing an n-variable Boolean circuit where all non-input nodes are NAND gates.
Question: Is there a truth assignment of the variables such that the value of the circuit is *true*?

Let us assume the following: $V = \{v_1, \ldots, v_m\}$, where the nodes $\{v_1, \ldots, v_n\}$ are the *input nodes*, the other nodes called *gates*, in particular v_m is the *output gate*. $E = \{e_1, \ldots, e_k\}$, such that for all $i = 1, 2, \ldots, k$, $e_i = (v_{s_i}, v_{t_i})$, where $s_i < t_i$. The two input nodes of a gate v_i are v_{l_i} and v_{r_i}.

We will use the initial set T_2^m. The string $s = p_1 a_{i_1} p_2 a_{i_2} \ldots p_m a_{i_m} \in T_2^m$ represents such an assignment of truth-values to the circuit G, where the jth input variable of the circuit ($j = 1, \ldots, n$) is *true* if $i_j = 2$, and *false* if $i_j = 1$. Similarly, a gate v_j ($j = n+1, \ldots, m$) is assigned to the output value of *true* if $i_j = 2$, and *false* if $i_j = 1$. Of course, such an assignment may not be consistent with the circuit. It is only consistent when for each $j = n+1, \ldots, m$, the truth-value assigned to the gate v_j equals to the result of the logical NAND operation on the values assigned to its input nodes, namely v_{l_j} and v_{r_j}. The algorithm CIRCUIT-SATISFIABILITY checks for this property gate by gate.

CIRCUIT-SATISFIABILITY

1 $Input(T = T_2^m)$
2 **for** $i \leftarrow n+1$ **to** m **do**
3 $Copy(T, \{T_{ff}, T_{ft}, T_{tf}, T_{tt}\})$
4 $Remove(T_{ff}, \{p_{l_i}a_2, p_{r_i}a_2, p_i a_1\})$
5 $Remove(T_{ft}, \{p_{l_i}a_2, p_{r_i}a_1, p_i a_1\})$
6 $Remove(T_{tf}, \{p_{l_i}a_1, p_{r_i}a_2, p_i a_1\})$
7 $Remove(T_{tt}, \{p_{l_i}a_1, p_{r_i}a_1, p_i a_2\})$
8 $Union(\{T_{ff}, T_{ft}, T_{tf}, T_{tt}\}, T)$
9 **end**
10 $Remove(T, \{p_m a_1\})$
11 Select(T)

The procedure separates the input for each gate into four distinct sets based on the values of its two input nodes. Strings where the output value of the examined gate does not agree with the computed output of the gate are removed. The strings that survive the above filtering are encodings of consistent truth

value assignments of the circuit. In step 10 strings are removed that encode such an assignment, where the output of the circuit is false. It is easy to see, that the remaining strings in T (if any) are encoding such truth assignments to the n input variables of the circuit, for which the circuit outputs true. Hence the *Select* operation in step 11 gives such an assignment, or if there is none, answers *empty*. Since all used operations can be performed in constant time, the time complexity of the algorithm is proportional to the number of gates in G. Please note that the above algorithm can easily be generalized to handle arbitrary circuits. □

4 Conclusions

In this work it was shown for a couple of NP-complete problems, that they can be efficiently solved in a theoretical model of molecular computing. Although laboratory experiments are needed to justify the practical use of the given algorithms, we believe, that these results do not have theoretical consequences only. Since the carefully designed Parallel filtering model uses only such operations, that are feasible in practice, we gain new proofs for the claim that practical molecular computing may outperform electronic computers.

References

[1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994. ftp://ftp.krl.caltech.edu/pub/users/brown/adleman.ps.gz.

[2] Richard J. Lipton. Speeding up computations via molecular biology. Unpublished manuscript Dec. 9, 1994, ftp://ftp.cs.princeton.edu/pub/people/rjl/bio.ps.

[3] Leonard M. Adleman. On constructing a molecular computer. volume 27 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996. http://citeseer.nj.nec.com/adleman95constructing.html.

[4] Diana Rooss and Klaus W. Wagner. On the power of DNA-computers. Technical Report 103, University of Würzburg, 1995. http://citeseer.nj.nec.com/55405.html.

[5] Eric Bach, Anne Condon, Elton Glaser, and Celena Tanguay. DNA models and algorithms for NP-complete problems. In *Proceedings, Eleventh Annual IEEE Conference on Computational Complexity*, pages 290–300, Philadelphia, Pennsylvania, 24–27 May 1996. IEEE Computer Society Press.

[6] Eric B. Baum and Dan Boneh. Running dynamic programming algorithms on a DNA computer. In *Proceedings of the Second Annual Meeting on DNA Based Computers, Princeton University*, 1996. http://citeseer.nj.nec.com/baum96running.html.

[7] Martyn Amos, Alan Gibbons, and David Hodgson. Error-resistant implementation of DNA computations. In *Proceedings of the Second Annual*

Meeting on DNA Based Computers, Princeton University, 1996.
http://citeseer.nj.nec.com/26034.html.

[8] Dan Boneh, Christopher Dunworth, and Jiří Sgall. On the computational power of DNA. *Discrete Applied Mathematics*, 71(1-3):79–94, 1996.
ftp.cs.princeton.edu/pub/people/dabo/biocircuit.ps.Z.

[9] Mitsunori Ogihara and Animesh Ray. Simulating boolean circuits on a DNA computer. Technical Report TR631, University of Rochester, 1996.
http://citeseer.nj.nec.com/ogihara96simulating.html.

[10] Richard Beigel and Bin Fu. Molecular computing, bounded nondeterminism, and efficient recursion. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, number 1256 in Lecture Notes in Computer Science, pages 816–826, 1997.
http://citeseer.nj.nec.com/beigel96molecular.html.

[11] Greg Gloor, Lila Kari, Michelle Gaasenbeek, and Sheng Yu. Towards a DNA solution to the Shortest Common Superstring Problem. In *Proceedings of IEEE'98 International Joint Symposia on Intelligence and Systems*, pages 111–113.
http://citeseer.nj.nec.com/498.html.

[12] Lila Kari, Greg Gloor, and Sheng Yu. Using DNA to solve the Bounded Post Correspondence Problem. *Theoretical Computer Science*, 231(2):193–203, January 2000.

[13] Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages*. Springer Verlag, Berlin, Heidelberg, New York., 1997.

[14] Martyn Amos. The complexity and viability of DNA computations (extended draft) CTAG-97001. The University of Liverpool, 1997.
http://citeseer.nj.nec.com/511.html.

[15] Martyn Amos. *DNA Computation*. PhD thesis, Department of Computer Science, University of Warwick, UK, 1997.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. W. H. Freeman & Co, 1979.

[17] Christos H. Papadimitrou. *Computational Complexity*. Addison–Wesley Publishing Company, 1994.