Technical Report No. 2006-523
Aspects of Biomolecular Computing*

Naya Nagy and Selim G. Akl
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6
Email: nagy,akl@cs.queensu.ca
October 12, 2006

ABSTRACT

This paper is intended as a survey of the state of the art of some branches of Biomolecular Computing. Biomolecular Computing aims to use biological hardware (*bioware*), rather than chips, to build a computer. We discuss the following three main research directions: DNA computing, membrane systems, and gene assembly in ciliates. DNA computing combines practical results together with theoretical algorithm design. Various search problems have been implemented using DNA strands. Membrane systems are a family of computational models inspired by the membrane structure of living cells. The process of gene assembly in ciliates has been formalized as an abstract computational model. Biomolecular Computing is a field in full development, with the promise of important results from the perspective of both Computer Science (models of computation) and Biology (understanding biological processes).

1. Introduction

Virtually all computers we currently use are based on semiconductor technology. Silicon semiconductors perform all computations in industry today. Although silicon based computers improve rapidly, computational needs also increase, as ever more complex problems can be formulated and wait to be solved. Searching for more powerful computers, opens the door to researching utterly different technologies, miniaturized to even elemental components (such as molecules or atoms). These are expected to challenge the limits of current semiconductor technology.

Many existing physical and chemical systems in our natural environment are surprising by their highly organized structure. None of them, however, can compete with the simplest *biological* system. Any biosystem excels in terms of its complexity, its design, the processes it performs, its adaptability to changing circumstances, and so on. A colony of bacteria, a cell, or a cell complex can be viewed and

*This research was supported by the Natural Sciences and Engineering Research Council of Canada studied as micro scale living (*in vivo*) biological systems. Subsequently, any chain of interdependent biochemical processes affecting synthesized organic molecules also forms a biological system (*in vitro*). In the latter case, organic molecules refer to molecules of DNA, RNA, proteins, enzymes, and so on, which are not alive themselves but play an active, indeed crucial, role in the life of a cell or body. For example, a set of DNA strands in a water solution undergo transformations of lengthening shortening, annealing, multiplication, and filtering. This yields a well defined in vitro biological system.

The idea follows immediately: Could a biological system (a collection of cells) work at a task defined by a human? Living organisms already accomplish vastly complex tasks in nature, and do so reliably.

The goal of *Biomolecular Computing* is exactly this: To solve computational problems using a biological system rather than a conventional computer [46]. When a biosystem yields the *"hardware"*, aptly named *"bioware"* a new *biocomputer* is born.

This survey presents results of Biomolecular Computing. It should be noted that the latter is one of several research domains in which Biology and Computer Science meet. Thus:

- The computational power of current computers is used to address biology problems in *Bioinformatics* and *Biomedical Computing*.
- The design of algorithms has benefited from biology knowledge, by using biological principles or copying biological processes. Several classes of so called biologically-inspired algorithms exist, such as *Genetic Algorithms*, *Evolutionary Algorithms*, *Ant Colony Algorithms*, and so on.
- Computers can be made of biological matter (DNA molecules, cells, cell colonies, etc). The computational models are directly defined by the bioware used and algorithms are designed to exploit these models. This is *Biomolecular Computing* and the object of this paper.

2. DNA Computing

In DNA computing the computation is performed by DNA strands. The strands are obtained synthetically. To execute a computation step means to apply some lab manipulation technique to the DNAs in a test tube. The strands act both as processors and memory units. Experiments using DNA computing have solved NP-complete problems in linear time, covering an exponential space in parallel. The strength of DNA computing is huge parallelism in a relatively small physical space, due to the molecular size of the computation units.

The DNA, Deoxyribonucleic Acid, is a long, linear molecule. It consists of a chain of nucleotides held together by phosphodiester bonds. Each nucleotide contains one of four bases: A (adenine), C (cytosine), T (thymine), and G (guanine). As the rest of the nucleotide is invariably the same (a phosphate and a sugar), the DNA chain, or strand, can be fully described by the sequence of bases. The bases A with T, and C with G respectively, are Watson-Crick complementary. This means that, complementary bases brought into proximity form hydrogen bonds and tend to remain together. Because of the hydrogen bond attraction, two strands with pairwise complementary bases, stick together forming a double helix. A double helix DNA molecule is also called a double stranded DNA molecule (*ds*DNA). The simple, one strand DNA is a single stranded DNA molecule (*ss*DNA).

2.1. Molecular Operations

Nowadays, DNA strands are commonly manipulated in the lab. Several standard techniques are used in DNA computing to perform a DNA computation. The following is a list of the most common DNA processes and techniques used in DNA computing.

1. **annealing and denaturation**. As already mentioned, complementary DNA strands tend to stick together forming *ds*DNA. This process is called *annealing*. *ds*DNAs are stable at low temperatures, lower than 35° C. When heated to 65° C, the hydrogen bonds are broken and the *ds*DNAs are separated into *ss*DNAs. This separation is called *denaturation*.

2. **repairing nicks**. A nick is a broken phosphodiester bond. Consider two *ss*DNA strands that both anneal to a longer complementary *ss*DNA strand. By annealing they are brought side by side, but their sugar-phosphate backbone link is missing. Nicks are repaired by an enzyme: DNA ligase. DNA ligase added to a DNA solution repairs broken nicks.

3. **multiplying DNA using polymerase chain reaction (PCR)**. PCR is a common method to create copies of *ds*DNA. The technique applies to short strands, up to 10 kb (kilo base pairs). One PCR cycle performs the following steps. The *ds*DNA strands are denaturated. *Primers* (small *ss* complementary to the ends of the DNA strand) are attached to the *ss*DNA. The enzyme DNA polymerase copies the rest of the strand. Hence the number of DNA strands is doubled. This cycle is repeated 20-35 times, which increases the quantity of DNA exponentially.

4. **measuring the length of DNA molecules using gel electrophoresis**. The *length* of a *ss* molecule is the number of nucleotides in the strand. This length can be determined using *gel electrophoresis*. DNA is an acid and therefore electrically charged. In an electric field, DNA molecules tend to migrate towards the positive electrode. The migration medium is a gel. The gel slows down the larger molecules, whereas small molecules travel faster. In fact, the distance traveled by a molecule in the gel is proportional to its size (length). Thus, the length of the molecule is determined by its position in the gel.

5. **fishing out known substrands using magnetic bead separation**. Given a solution with various different strands of DNA, magnetic bead separation extracts the molecules that contain a certain substrand. The method uses the annealing property of complementary strands. The complement of the substrand of interest is attached to a magnetic bead. This complement is called the *probe*. Under proper temperature conditions, a great majority of

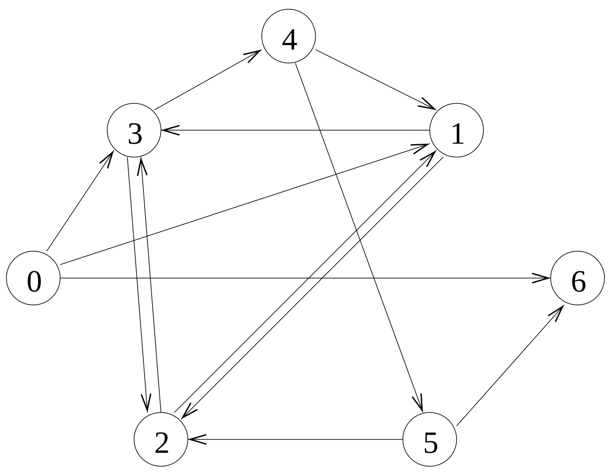

Figure 1: The graph in Adleman's experiment.

target molecules will attach to the probes on the beads. The beads are then taken out of the solution by an electric field.

6. **reading out the sequence of nucleotides**. The DNA sequencing method designed by Fred Sanger (around 1970) determines the sequence of the four bases in a *ss* DNA. New complementary DNA single strands are synthesized from the original by the action of polymerase. Sanger's method terminates the copied molecules at all intermediate lengths. This is done by chemically altered nucleotides (dideoxynucleotides). The sugar in these nucleotides lacks the hydroxyl group and thus cannot further extend the phosphodiester bond. Reading the different lengths determines the position of the modified nucleotides and consequently their bases.

2.2. Adleman's Experiment

DNA computing was pioneered by an experiment believed to be the first computer built from DNA molecules. Leonard Adleman [2], in 1994, showed that manipulating DNA is equivalent to running the implementation of an *abstract* computation. He applied the well established DNA properties and manipulation techniques described above to perform the computation. It turns out that using these simple techniques is enough to solve the well known NP-complete Hamiltonian path problem. A Hamiltonian path in a graph is a path starting in a designated source vertex, ending in a designated destination vertex, and visiting all other vertices exactly once. The Hamiltonian path problem means to search for a Hamiltonian path in a graph. For a graph with n vertices, no algorithm is known for solving the Hamiltonian path problem in time linear in n. The only known solution is to enumerate all possible paths, which requires $O(n!)$ time.

Adleman's experiment effectively solved the Hamiltonian path problem on a small instance of a graph. The graph is given in fig. 1. It contains seven vertices and the goal is to find a Hamiltonian path starting in vertex 0 and ending in vertex 6. It can be verified visually that the only Hamiltonian path in fig. 1 is $0-1-2-3-4-5-6$.

The algorithm is an exhaustive search of all paths in the graph. Each DNA molecule represents one path. The algorithm [3] is short and elegant:

Given a graph with n vertices,

1. Generate a set of random paths through the graph.
2. For each path in the set:
   (a) Check whether that path starts at the start vertex and ends with the end vertex. If not remove that path from the set.
   (b) Check whether that path passes through exactly n vertices. If not, remove that path from the set.
   (c) For each vertex, check if that path passes through that vertex. If not, remove that path from the set.
3. If the set is not empty, then report that there is a Hamiltonian path. If the set is empty, report that there is no Hamiltonian path.

It remains to explore how DNA can implement the simple algorithm described above.

The first step is easily solved, if the input (the graph) is encoded in an advantageous way. The graph, vertices and edges, have to be translated to DNA strands. This is the point where Watson-Crick complementarity gets fully exploited. Each primitive, vertex or edge, is encoded in a *ss* DNA of twenty nucleotides. Consider for the graph in fig. 1, that the seven vertices are encoded by some arbitrary, distinct *ss* DNA strands, of length 20 each. We denote that sequence of nucleotides for vertex i: $n_{i,1}n_{i,2}n_{i,3}...n_{i,20}$. In particular, the strands $n_{0,1}n_{0,2}n_{0,3}...n_{0,20}$ and $n_{1,1}n_{1,2}n_{1,3}...n_{1,20}$ represent the vertices 0 and 1 respectively. The edges are encoded depending on the already existing vertices' strands. Let nucleotide $\overline{n}$ be the complement of nucleotide n. The strand representing the directed edge (i,j) is constructed as follows: the first half (10 nucleotides) of the edge are the complement of the second half of vertex i; the second half (next 10 nucleotides) of the edge are the complement of the first half of vertex j. Therefore, the edge (i,j) is represented by the *ss* DNA $\overline{n}_{i,11}\overline{n}_{i,12}...\overline{n}_{i,20}\overline{n}_{j,1}\overline{n}_{j,2}...\overline{n}_{j,10}$. In particular, for the edge $(0,1)$ in fig. 1 the strand is $\overline{n}_{0,11}\overline{n}_{0,12}...\overline{n}_{0,20}\overline{n}_{1,1}\overline{n}_{1,2}...\overline{n}_{1,10}$.

The advantage of this encoding can be seen immediately. Vertex 0 anneals to edge $(0,1)$ (fig. 2) and the second half of edge $(0,1)$ anneals to vertex 1 (fig. 3). Vertex 1 has been appended to vertex 0. DNA ligase, the enzyme specialized in repairing nicks, takes care to bind the two strands by the necessary phosphodiester bond. The new single strand $0-1$ containing vertex 0 and vertex 1 shows that there exists a path from 0 to 1.

The initial test tube is a water solution containing many copies of each vertex and edge of the graph. When the test tube is shaken, vertices anneal to edges

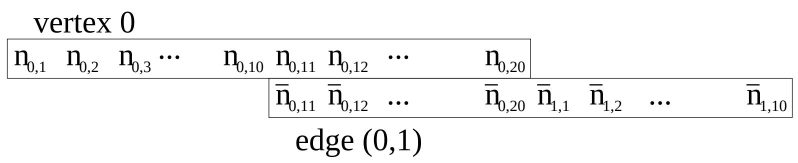
vertex 0
$n_{0,1}$ $n_{0,2}$ $n_{0,3}$ ··· $n_{0,10}$ $n_{0,11}$ $n_{0,12}$ ··· $n_{0,20}$
$\bar{n}_{0,11}$ $\bar{n}_{0,12}$ ... $\bar{n}_{0,20}$ $\bar{n}_{1,1}$ $\bar{n}_{1,2}$ ... $\bar{n}_{1,10}$
edge (0,1)

Figure 2: Vertex 0 annealed to edge $(0, 1)$.

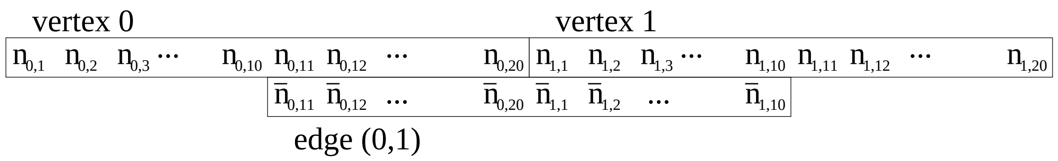
vertex 0
vertex 1
$n_{0,1}$ $n_{0,2}$ $n_{0,3}$ ··· $n_{0,10}$ $n_{0,11}$ $n_{0,12}$ ··· $n_{0,20}$ $n_{1,1}$ $n_{1,2}$ $n_{1,3}$ ··· $n_{1,10}$ $n_{1,11}$ $n_{1,12}$ ··· $n_{1,20}$
$\bar{n}_{0,11}$ $\bar{n}_{0,12}$ ... $\bar{n}_{0,20}$ $\bar{n}_{1,1}$ $\bar{n}_{1,2}$ ... $\bar{n}_{1,10}$
edge (0,1)

Figure 3: Vertex 1 appended to vertex 0 by annealing to edge $(0, 1)$.

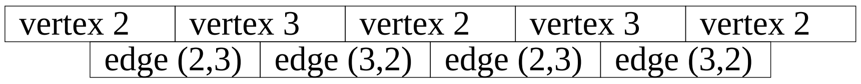

Figure 4: The path $2 - 3 - 2 - 3 - 2$.

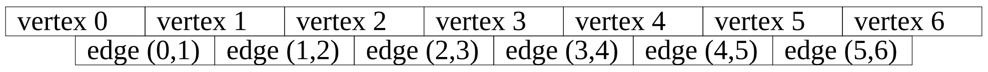

Figure 5: The Hamiltonian path $0 - 1 - 2 - 3 - 4 - 5 - 6$.

forming valid paths in the graph. Fig. 4 shows the path $2-3-2-3-2$, fig. 5 shows the Hamiltonian path $0-1-2-3-4-5-6$. If there are sufficiently many copies of each primitive, the probability that a particular path (here the Hamiltonian path) has been formed is close to 1.

Step 2 is done in parallel on all paths-strands in the test tube. Paths starting with the source 0 and ending with the destination 6, step 2a, are weeded out using PCR. Paths of the length of 7 vertices are separated using gel filtering: the gel corresponding to $7 \times 20 = 140$ nucleotides is cut out and used further. Step 2c loops over all vertices in the graph (excluding the source and destination). For each vertex, all paths that contain that vertex are extracted by magnetic bead separation and used further. After step 2, only Hamiltonian paths remain in the test tube: paths of length 7, visiting all vertices.

In step 3, the DNA sequence of the strand left in the tube, if any, is read out using Sanger's sequencing method.

The algorithm described above is in the category of exhaustive search algorithms. The search space for finding the Hamiltonian path is the set of all possible paths in the graph. This space is inspected in parallel. All paths are tested for Hamiltonian validity at once. Although, Adleman spent seven days in the lab to perform the molecular operations manually, the number of computation steps required by the algorithm is linear in the number of vertices of the graph. This result is remarkable, since the best sequential algorithm known needs exponential time. The polynomial-time complexity of Adleman's implementation is accomplished by the massive parallelism that computations using DNA strands can achieve. Every DNA strand represents one path in the search space. As one molecule represents one path, a test tube has the potential of holding a huge number of paths. $10^{15} - 10^{17}$ strands of DNA are routine in a small test tube experiment [45]. In fact, the search space is exponential in the number of vertices. Thus, the algorithm works in linear time, but needs to cover an exponential space. This space can be inspected in linear time due to the exponential parallelism of the algorithm. Note that, there is no constant bound on the number of threads to be executed in parallel. The parallelism depends on the size of the problem, in particular the number of threads executed in parallel is exponential in the number of vertices.

Note that this algorithm is not perfect [3]. The Hamiltonian path is not guaranteed to be generated in the first step. There is a degree of nondeterminism in generating the paths. An edge (i, j) may stick to *any* occurrence of the vertex i and respectively the vertex j, possibly favoring some paths. If the paths in step 1 are generated randomly enough and if the set of initial primitives is large enough, then the probability is high that the algorithm will give a correct answer.

3. Exploiting Parallelism

The prospect of having a DNA computer built, means to have regular and easy access to performing $10^{15} - 10^{17}$ operations in parallel. This number is huge compared to what can reasonably be expected from conventional parallel computers. It also has the potential of sweeping large search spaces at once. Pioneering with Adleman's contribution, exhaustive search algorithms, based on DNA computing, have been designed for several NP-complete problems. Adleman's exhaustive search procedure does not necessarily apply as such to the other NP-complete problems, as the translation into DNA operations is not obvious or straightforward.

Lipton [35] solves the well-known satisfiability problem (SAT) using the same algorithmic steps. Consider a boolean expression that contains only variables, the connectives AND, OR, and NOT, and parentheses. An assignment of TRUE or FALSE to all variables, that makes the entire boolean expression true, is an assignment that *satisfies* the expression. The SAT problem is a decision problem, it aims to determine whether there exists an assignment that satisfies a given boolean expression. Lipton's algorithm finds such an assignment, if it exists, or gives a negative answer otherwise. The ingenuity of the algorithm consists in translating the boolean formula into a graph. The paper considers boolean formulas of the following types:

1. a conjunction of m clauses, where a clause is a disjunction of a fixed number of literals (boolean variables or negated boolean variables).

2. any boolean formula containing **negation**, **and**, **or**, and **parentheses**.

In both sub-cases, the DNA computer works in linear time in the length of the boolean formula.

As there is an equivalence between general boolean formulas and contact networks [49], these networks also find their solution in Adleman's exhaustive search [35]. Contact networks are oriented graphs with a designated source and destination vertex. Edges are assigned boolean variables or their negation. The source is *connected* to the destination, if there exists an assignment to the variables, such that all edges on the connecting path are true.

Chang et al. [15] design an algorithm to solve the maximum independent set problem. The algorithm finds the maximum sized subset V' of vertices in a graph, such that no two vertices in V' are connected by an edge. The execution time is linear in the number of edges.

Ouyang et al. [37] solve the maximum clique problem and also manage a practical implementation for a graph with 6 vertices. The maximum clique is a subset of vertices of maximum size such that the resulting subgraph defined by these vertices is fully connected. The algorithm is linear in the number of edges of the complementary graph.

A different algorithmic solution to the Hamiltonian path problem is found in Beaver's paper [8]. Although the general idea is the same, namely parallel exhaustive search of an exponential space, the procedure is different. A text insertion/deletion method is used to test for visiting every node. This method temporarily lengthens the DNA strands and selects correct strands according to their length. The final reading out of the path first reduces the number of Hamiltonian paths existing in the final water solution. This is useful as standard sequencing techniques work only on homogeneous solutions. The number of paths is reduced by selecting the path that visits the smallest numbered vertex. Thus, this method is better suited for Hamiltonian path problems with multiple solution paths.

Boneh et al. [12] designed an algorithm for breaking the Data Encryption Standard (DES). This is a real life problem. DES is an encryption procedure. It encrypts

64 bit messages with a 56 bit key. The algorithm finds the secret key by searching all keys exhaustively. The key that produces the desired encryption is separated from the rest. For a chosen plain-text attack, with the preparation in advance of the solution containing all key encodings, DES could be broken in one day. The algorithm assumes error free molecule operations. This is unrealistic for the size of the problem (2^{56} keys).

4. Practical Considerations

Building a DNA computer remains an open challenge. Adleman's experiment stands as the first instance of a DNA computer ever built in reality. The method has been confirmed by Kaplan [30] who repeated the experiment.

Several research groups have implemented small instances of the SAT problem. All implementations are exhaustive search algorithms. Their execution time is polynomial in the number of variables and they search an exponential space in parallel.

Liu et al. [36] implemented a 3-SAT problem (clauses contain three variables at most) with four variables (64 solution candidates). They used surface-based chemistry. The set of solution candidates (DNA molecules) after being synthesized, are attached to a surface. The testing of the candidates is done in cycles, each cycle tests one clause. In each cycle, candidates that fail the test are removed from the surface by an endonuclease enzyme. The method has the potential of being automated. Unfortunately, the molecular operations, including reading out the final result have quite high error rates (approx. 4% per operation). These affect the scalability of the solution. Another implementation with DNA of a 4-variable SAT is found in Yoshida and Suyama [53]. Here the search procedure is breadth first.

The paper by Faulhammer et al. [21] implicitly solves the SAT problem in their implementation of the chess "Knight problem". Given an $n \times n$ chess-board, the knights are to be placed on the board so that no knight is attacking any other. The no-attack constraint is expressed as a boolean formula. A successful knight configuration is equivalent to a truth assignment satisfying the boolean formula. The Knight problem was practically solved for $n = 3$ and the algorithm runs on RNA molecules rather than DNA. The algorithm searches exhaustively a space of $1,024$ solution candidates.

Though still an exhaustive search algorithm, Sakamoto et al. [47] implement the 3-SAT problem in an *autonomous* manner. *Autonomous molecular computations*, described by Hagiya [26], consist of a succession of autonomous molecular reactions. The first step in Adleman's experiment, generating the paths in the graph, is considered an autonomous computation step. The paths are formed autonomously by annealing reactions. These reactions are based on the DNA molecular structure only and do not need any further control from the outside. For the 3-SAT problem, the logical constraints have to be encoded *into* the DNA sequence. Thus, the algorithmic steps do not depend on the particular instance of the 3-SAT formula to be checked. The algorithm performs exactly the same steps for any boolean formula to be checked for satisfiability. It is only the initial data, the initial pool of DNA strands, that depend on the specific boolean formula. All other implementations of the SAT problem (including Adleman's Hamiltonian path implementation) are not autonomous. The autonomous procedure presented in [47] is based on the property of *ss*DNA strands to form hairpins, if two long enough subsequences are complementary along the strand. These hairpin forming strands are erroneous solutions, that assign *true* to both a variable a and its negation (complement) $\neg a$. Hairpin strands are removed from the solution by enzymatic digestion. The success rate of this approach is lower than that of implementations using common molecular biology techniques. Nevertheless, the algorithm does not depend on the length of the formula, nor does it depend on the number of variables and is therefore a constant time algorithm. This is a major difference. The implementation used a 6-variable, 10-clause instance on the 3-SAT. The algorithm distinguished among $3^{10} = 59,049$ candidate solutions from the initial test tube.

The largest implementation of Lipton's solution to the SAT problem can be found in Braich et al. [13]. This implementation solved the 3-SAT problem for 20 variables. The algorithm searched a space of $1,048,576$ possible truth assignments. It used a simplified version of the sticker model. The sticker model or system will be defined in section . The model relies on DNA's annealing property. The only operation used in Braich's implementation is separation based on a subsequence. The experiment performed 24 separation operation. Error analysis reveals a 0.87 probability of a correct strand surviving a separation step. The probability of 1 incorrect strand being accidentally retained in one step is at most 3.75×10^{-4}. The authors suggest that the size of the problem can be increased by using periodic PCR amplifications for error correction [11].

With the intention of extending the applicability of the DNA computer, Guarnieri et al. [23] implement a method to add nonnegative numbers. The procedure computes each bit of the sum iteratively, using the property of DNA polymerase to extend a DNA strand starting from a primer. The length of the DNA strand holding the result is proportional to the value of the numbers added. Practical results are presented for adding two binary digits. The algorithm shows the capacity of DNA to add, but it falls short of two desirable characteristics. First, the implementation does not allow for iterative addition of several numbers, the algorithm adds exactly two numbers and the result cannot be reused as further input. Second, addition cannot be performed in parallel, thus the basic advantage of DNA computing is not exploited.

4.1. Dealing with Errors

Unfortunately, DNA manipulations are error prone. Especially in view of practical algorithms with thousands of lines, errors accumulate, rendering results useless.

Boneh et al. [11] analyze two sources of errors in exhaustive search algorithms. First, the extraction operation (usually bead separation) may fail (typically by 5%) to extract all matching DNA strands or it may mistakenly extract faulty strands (again typically 10^{-4}%). Secondly, DNA strands decay in time, they have a *half-life* in the order of weeks. Both sources of error, naturally decrease the probability to have a solution strand in the final test tube. To remedy this shortcoming, the paper proposes applying PCR every several steps. Thus, the survival probability of the solution strands can be increased from virtually 0 to over 50 %. This method can be applied to decreasing volume algorithms and turns them into constant volume algorithms. In addition, the paper offers a novel method (the double encoding scheme), subject to laboratory testing, to improve bead separation. The idea is to encode the separation templates two times, rather than once, along the strands in the search space. This means that, matching strands have now two sites to attach to the bead, thus hopefully increasing their extraction probability.

Ouyang et al. [37] effectively computed the maximal clique of a graph with six vertices. They depicted two sources of error with which they practically had to deal with. First, PCR produces some single stranded DNA, rather than double strands, which cannot be cut by restriction enzymes. The remedy is to digest (destroy) the *ss*DNA. The second source comes from incomplete splicing by the restriction enzyme. Repeating the digestion-PCR process reduces the ratio of uncut strands. On the other hand, point mutations are tolerated by the algorithm. The paper also considers improving DNA manipulation: using different polymerases and using an automatic device to accelerate readouts.

Shah et al. [48] describe a potential experiment to find a substring pattern along a single DNA strand. It finds the number of occurrences of the pattern together with their position in the strand.

4.2. Is DNA the only option?

Stuart Kurtz et al. [33] talk about dropping DNA altogether, in the quest for a more feasible molecule. Unfortunately only in a theoretical stage, they propose a molecule that is a generalization of DNA, RNA, and protein. They claim that such a molecule, the *computational nucleic acid* (CNA), is indeed chemically feasible. CNA would replicate like DNA. Even more surprising, a mechanism similar to protein transcription would allow the reading out of a CNA molecule and then based on its encoding, a new CNA is created. This would be a computation step. Nevertheless, CNA still waits to be synthesized, before any attempt on error evaluation of the computation steps can be made.

5. The Theoretical Models

How much can a DNA computer do? Is it theoretically at least as good as a conventional computer?

The operations a DNA computer performs are defined by the DNA manipulation techniques used. A strand of DNA represents the physical support for a string. DNA operations translate into string operations. A whole variety of mathematical models [44] have been formalized depending on the set of DNA operations the model allows.

5.1. The Sticker System

The mathematical model most faithful to Adleman's experiment is the *sticker system.* Here, computational data are encoded in double stranded (*ds*) DNA. These *ds* blocks of DNA, also called *dominoes*, can have single stranded (*ss*) overhangs on either or both ends (see fig. 6). The sticker system has only one generic operation: the sticking operation. Two *ds* DNA strands can stick together forming one, longer *ds* strand, provided their ends match according to the Watson-Crick

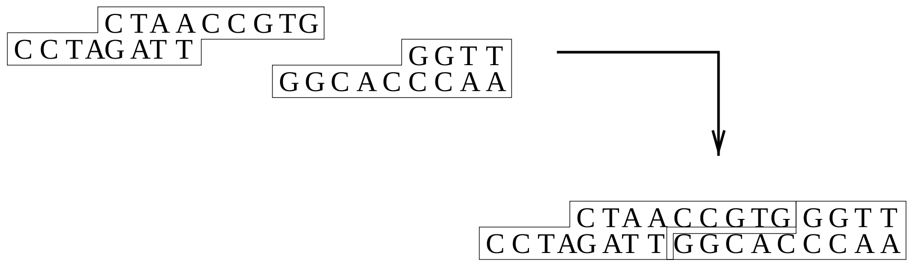

Figure 6: The operation of sticking.

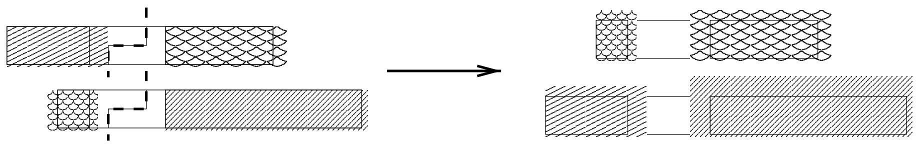

Figure 7: The splicing operation.

complementarity rule.

This system, though very simple in its definition, turns out to be complex enough. In fact, the generative capacity of the sticker system [31] is in the category of recursively enumerable languages. The expressive power of the sticker system is equivalent to the Turing Machine.

5.2. The Splicing System

Based on a different DNA technique is the splicing system. Splicing means recombining two *ds* DNA strands after cutting them at a certain site (see fig. 7). Endonuclease enzymes are specialized in cutting the DNA at a site determined by a specific sequence of oligonucleotides. Head [29] and Păun [41] have defined a formal computability model based on the splicing operation. In [42], the generative power of the splicing system is investigated. The system is equivalent to the Turing Machine.

6. Technical Limitations

Our understanding of the life process at the molecular level is necessary to appreciate and use the bioware in our "computer". Molecular biology and cell biology give clear explanations to the following: The replication of DNA, the replication of RNA, and the synthesis of a protein from its gene through translation and transcription [34]. These are potential building blocks of computational units of biocomputers. Phenomena like genome remodeling [10] and RNA editing [50] are uncontrollable and partially understood and as such are inappropriate to be used in this field.

For decades, the Turing Machine (TM) has conceptually been linked to what we accept to be a computation. The Turing Machine initially has the input on its tape; it performs a series of state transitions and produces the solution on a (dedicated) output tape. In this classical view, computing with a biological system means to direct the biological or chemical development of the system from an initial state to an intended final state. The initial state represents the input of the problem encoded in some molecules. The final state contains the solution to the problem, retrievable from the structure of some molecule.

Nevertheless, current research formulates paradigms that challenge the TM model. Simple theoretical abstractions of potential applicative interest have been shown to require unconventional approaches. Models for which parallelism is inherent in their constitution exhibit computational transitions that a TM cannot simulate. Such problems have been described in geometry [6] (constraint driven transformations), in dynamic settings [5] (time varying data), in system control [7] (preventing a system from entering a chaotic, uncontrollable state).

The physical realization of a biocomputer promises processing units of molecular size. A DNA molecule acts like a processor or memory unit. In both cases, the size[†] is orders of magnitude smaller than for conventional computers. As such, huge parallelism can be obtained in a biocomputer of small size. Therefore, biological computational models can address inherently parallel computations.

Some advantages of using bioware have been found from the very beginning. Adleman [3] talks about the molecular size of the processing units, about the huge parallelism obtained in a common test tube, and the efficiency of ligation operations in terms of energy. The table below [4] compares characteristics of the biocomputer to the conventional computers today.

	Conventional Computer	*DNA Computer*
Storage (space for one bit 0 or 1)	10^{12} nm^3	1 nm^3
Speed (millions of instructions per second)	10^3	10^{14} (test tube full)
Energy (number of operations per joule)	10^9	2×10^{19}

[†]The mass of a DNA strand with n nucleotides is computed by the following formulas: $304 \times n + 79$ a.m.u. (atomic mass units) for a single stranded DNA molecule and $607 \times n + 158$ a.m.u. for a double stranded DNA molecule.
Thus, the weight of a single stranded DNA with 20 nucleotides is $304 \times 20 + 79 \approx 6000$. This means that $1g$ contains 10^{20} DNA molecules.

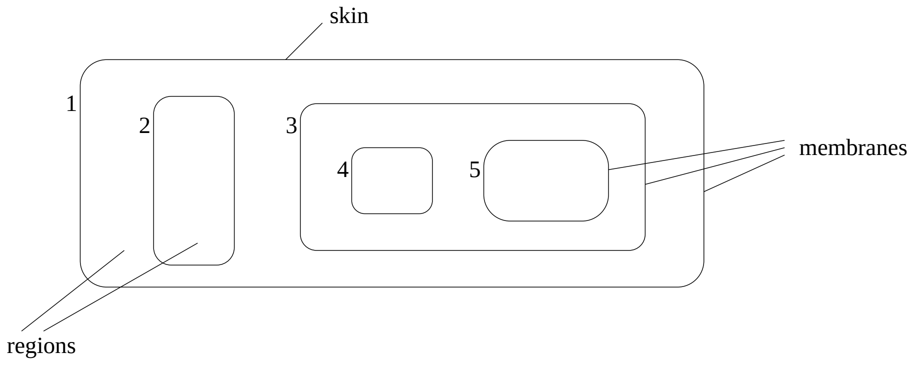

Figure 8: A system of membranes.

7. Membrane Computing

With *membrane computing* [40], we enter the realm of biologically *inspired* computation models. A biological system acts as a computer. For example, the cell is well studied in terms of its structure, life, and transformations in time. Then, based on some cell characteristics, a mathematical model is derived. The mathematical model mimics some processes of the cell. This model becomes the new computational model and is aimed to be able to perform computations, hopefully with a better performance than conventional models.

Note that the field of *biology inspired computing models* aims to create new *models* of computation. Such a model would naturally compete with established conventional models, like the Turing Machine. The field is not concerned with every form of computation inspired from biology. In particular, it does not model biological processes as algorithms or applications in general. Genetic and evolutionary algorithms [20,52] and Neural Computing [51] are not included in biology inspired computing models.

The founder of membrane systems, Gheorghe Păun [40], was inspired by the membranes surrounding cells and cell organelles. Cells have an outer skin membrane. Cell components are also identified by a separating membrane: the nucleus, the Golgi Apparatus (for protein processing), mitochondria, and various vesicles. Păun designed an abstract computational model using a hierarchy of membranes (fig. 8). Each membrane separates a region. Membranes can be created and destroyed during the computation. When a membrane is created, a new region is also created. When a membrane is destroyed, two regions are fused together.

According to the biological analogy, chemical molecules are generally contained by a certain membrane region. They do not freely migrate among membrane regions. The membrane can be completely impenetrable for certain molecules, or else, some molecules can penetrate a membrane according to certain restrictive rules. Nevertheless, chemicals of the same region can react / interact freely with any other chemicals of the region. The mathematical equivalent of molecules are symbols (a, b, c, ...). Membrane regions are multi sets of such symbols (ex. $a^2b^5c^3$ - meaning 2 copies of a, 5 copies of b, and 3 copies of c).

Chemical reactions are translated into production rules. For example, $ca \to cv$, means the element a is transformed into the element v in the presence of the catalyst c. Production rules are region specific.

The membrane system evolves by applying its production rules. All membrane regions evolve simultaneously, according to a global clock. In each time unit, in each region, all rules that can be applied, are applied nondeterministically, in a maximally parallel manner. The system halts when no further rules can be applied. The output is read either from the environment (outside the skin membrane) or as the content of some nondestructible membrane.

7.1. Variants of membrane systems

The basic system as described above allows for many extensions.

Evolution-communication P systems [14] rely on the property of biological membranes to act like filters. They make the passage of certain substances possible, but are impermeable for others. The mechanism of coupling molecules allows two molecules to pass a membrane as long as they pass it together, using a specific protein channel. In a *symport* transition, the molecules cross the membrane in the same direction, while an *antiport* transition requires the molecules to cross in opposite directions. Formally, (ab, in) or (ab, out) are symport rules, and $(a, out; b, in)$ is an antiport rule. It turns out that using *only* communication rules, the system is as powerful as the Turing machine [22]. Colson et al. [16] consider indicators of the form in_j, where j is a membrane label. Hence, objects can be *teleported* at any distance in the membrane structure.

P systems with active membranes allow the membrane structure to evolve dynamically during the computation. Besides the basic rules by which objects evolve and move in the membrane structure, a rule can dissolve a membrane or it can divide an elementary membrane in two. An interesting characteristic of these systems is that they can increase the number of membranes exponentially in linear time. This has been used to design algorithms for NP-complete problems in polynomial or even linear time, using exponential space.

In the basic P system, objects are atomic, denoted by letters (a, b, etc.). Extensions of membrane systems are obtained by simply considering structured objects. As such, a membrane may contain multisets of strings, trees, arrays, etc. For strings, depending on the rules, the system can be *rewriting P systems* or *splicing P systems.* An important property yields string rules of *rewriting with replication* [32]. These rules have the form: $a \leftarrow (u_1, tar_1)||(u_2, tar_2)||...||(u_n, tar_n)$, which means that a string x_1ax_2 is rewritten into n strings $x_1u_1x_2$, $x_1u_2x_2$, ..., $x_1u_nx_2$ into their destination regions tar_1, tar_2, ..., tar_n. Thus, exponential space can be obtained again in linear time.

In the basic model, the membrane structure is a treelike hierarchy. *Tissue-like P systems* define a different structure. The idea comes from tissues built up of cells. These cells communicate, if adjacent, via protein channels. Equivalently, membranes (in this case cells) are nodes in a graph. Connected cells communicate via symport and antiport rules. An even more general system is given by *population P systems* [9]. This system aims to model skin tissue, populations of bacteria, and colonies of ants. It allows cells to communicate, proliferate, die, change their characteristics, and behavior. Daughter cells inherit the links with the neighboring cells of the parent.

Comprehensive information about P systems can be found on the web [1].

7.2. Using Membrane Systems for Computationally Hard Problems

We have seen a few membrane systems with the capability of creating exponential space in polynomial (linear) time: P systems with membrane division, membrane creation, or string replication.

Algorithms have been designed for NP-complete problems. They run in polynomial time, with the tradeoff of exponential space. In particular, algorithms exist for

1. the SAT problem [43,54].
2. the Hamiltonian path problem [54].
3. the subset-sum problem [24]. Given a finite set A, a weight function $w : A \rightarrow N$ and a constant $k \in N$, determine whether or not there exists a subset $B \subseteq A$ such that $w(B) = k$.
4. the knapsack problem [38].
5. the QSAT problem [25]. Given a boolean formula $\varphi(x_1, x_2, ..., x_n)$ in conjuctive normal form, determine whether or not the (existential) fully quantified formula $\varphi^* = \exists x_1 \forall x_2 ... Q_n x_n \varphi(x_1, ..., x_n)$, where Q_n is $\exists$ for n odd and $\forall$ for n even, is satisfiable. The QSAT problem is actually a PSPACE-complete problem. Solving QSAT on a Turing machine requires polynomial space in the number of boolean variables with no time restrictions.

8. Gene Assembly in Ciliates

In the previous sections, we have presented computation systems that either used *bioware* as the basic technology, as in *DNA computing*, or were based on theoretical models inspired from biology, as in *membrane computing.* In both systems, one characteristic, so obvious that it goes unnoticed, is that they are human designed. *Gene assembly* on the other hand is a computing device naturally used by ciliates. Genes can be viewed as strings. As such, gene assembly basically is a mechanism to compute the original string from a scrambled version. The unscrambling process is formalized by only four simple operations. Nevertheless, these four operations are able to unscramble any permutation of the original string. A monograph of gene assembly in ciliates is found in [18].

Ciliates are unicellular organisms. Unlike most other organism of the kind, they have two functionally different nuclei: the micronucleus and the macronucleus. The macronucleus is genetically active during the normal activity of the cell. It provides the genes to be transcribed into messenger RNA and then be translated into proteins. The micronucleus is active only during sexual cell mating. Under adverse circumstances, mainly lack of food, a colony of ciliates may decide towards

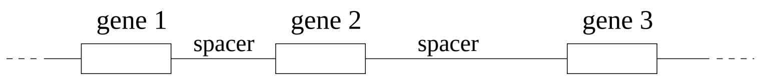

Figure 9: Genes along the chromosomal DNA of the micronucleus.

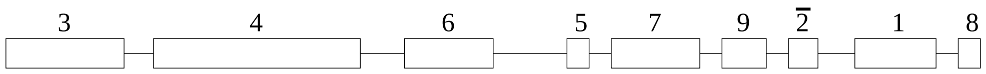

Figure 10: The micronuclear gene encoding actin in *Sterkiella nova*.

cell mating, rather than cell division. This is in the hope of producing genetically more resistant offspring. During cell mating a new macronucleus is formed using the genetic material from two different micronuclei. The final genes, though, belonging to the macronucleus are put together after a complex process of gene assembly.

In the micronucleus, the DNA chromosomial molecule is very long [28]. There are long sequences of nongenetic DNA between genes (see fig. 9). Each gene, in turn, is split into several segments called *macronuclear destined segments*, or MDS. The space between two MDSs are noncoding segments, called *internally eliminated segments*, or IES. Moreover, the MDSs do not necessarily appear in the order of the final gene, but are permuted in some apparently arbitrary way. Fig. 10 shows the micronuclear gene encoding *actin* protein in the bacterium *Sterkiella nova*. *Actin* has 9 encoding sequences, MDSs, and they come scrambled: $3, 4, 6, 5, 9, \overline{2}, 1, 8$. Note that MDS_2 is written out backwards in the micronucleus, shown as $\overline{2}$ in the figure. These sequences have to be rearranged in the canonic order for the macronucleus: $1, 2, 3, 4, 5, 6, 7, 8, 9$.

During gene assembly, 25000 genes are put together. The (new) macronucleus contains short strands of DNA, one gene per strand. Each gene-sized strand is replicated to hundred of thousands of copies.

Pointers have the mission to show the correct sequence of MDSs. Genes are assembled by homologous recombination of pointers. Suppose MDS_{i+1} is to follow MDS_i in the assembled gene. There exists a small sequence of 3 to 20 nucleotides, called *pointer*, at the beginning of MDS_{i+1}; this sequence is identical to the end of MDS_i (fig. 11). These pointers show where two contiguous MDSs have to meet. When MDS_{i+1} is assembled to MDS_i, the pointers are staggeredly cut by restriction enzyme and reassembled such that only one copy of the pointer exists in

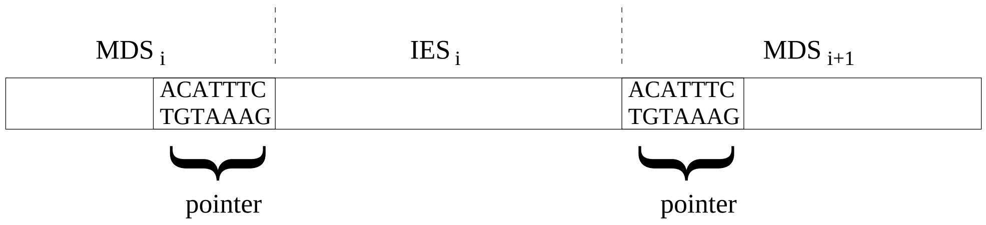

Figure 11: Pointer showing where recombination has to take place.

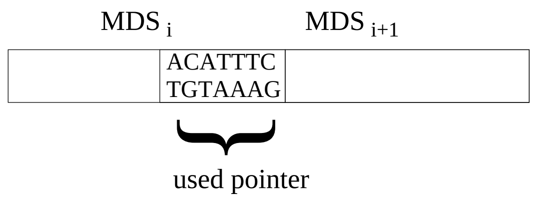

Figure 12: Two contiguous MDSs after their pointer has been used.

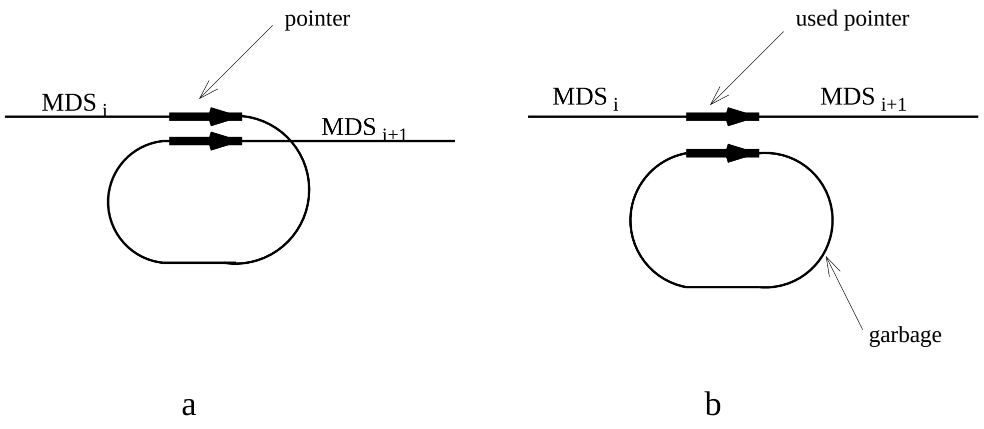

Figure 13: loop recombination with direct repeat: (a) before operation. (b) after the operation is completed.

the final gene (fig. 12). Once a pointer has been used, it ceases to act as a pointer any more.

8.1. Molecular operations

Although gene MDSs can come scrambled in any way, actually only a small set of *operation types* [39] is necessary to convey the capability of processing the canonic form from any arbitrary scrambling. The gene assemble process uses three operations: loop recombination (**ld**), hairpin recombination (**hi**), and double loop recombination (**dlad**).

1. **ld** - loop recombination - The pair of pointers marking this operation lie with the same orientation on the DNA strand (fig. 13 (a)). The sequence between the two pointers is excised and thrown away. The shortened, useful DNA strand contains exactly one copy of the pointer (fig. 13 (b)).

2. **hi** - hairpin recombination - The pair of pointers, marking this operation, lie in opposite directions (inverted) on the DNA strand (fig. 14 (a)). The sequence between the two pointers appears inverted after the operation (fig. 14 (b)). Both copies of the pointer sequence are retained by the final strand, though they do not act as pointers any further.

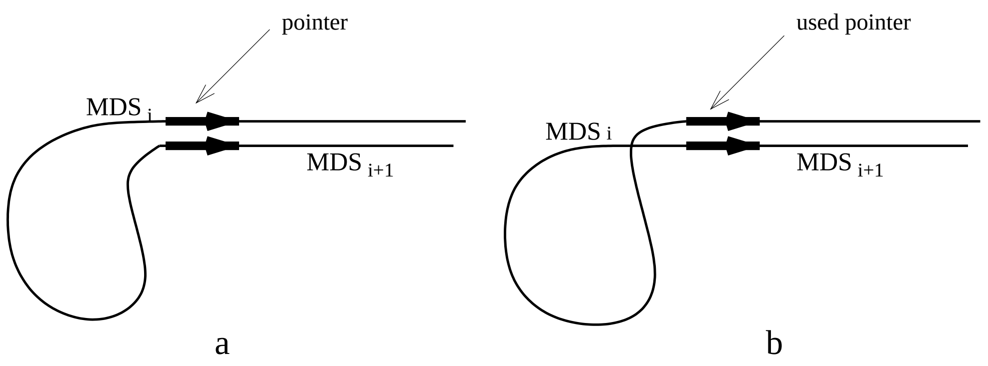

Figure 14: hairpin recombination with inverted repeat: (a) before operation. (b) after the operation is completed.

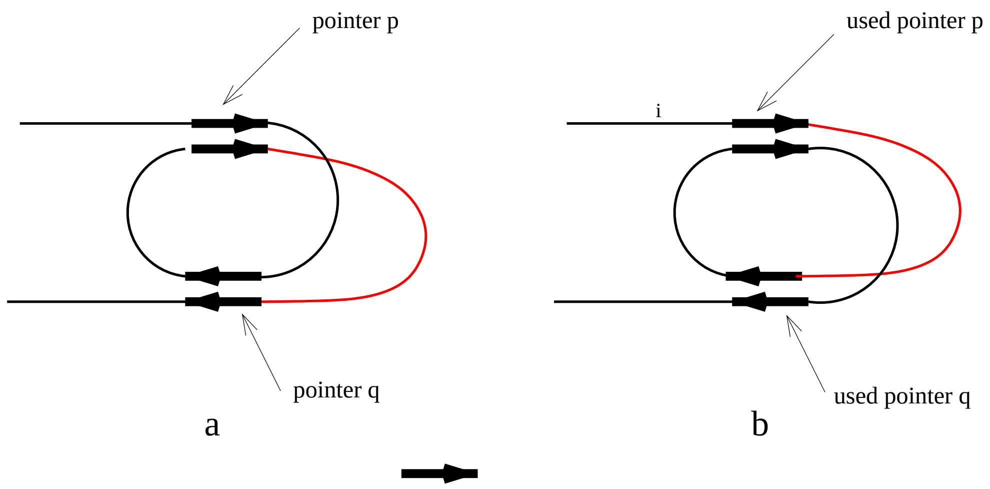

Figure 15: double loop recombination with direct pointers: (a) before operation. (b) after the operation is completed.

3. **dlad** - double loop recombination - This operation is defined by two pairs of pointers. They appear alternatively (fig. 15 (a)) on the DNA strand. The two sequences surrounded by different pointers are switched and their direction remains unchanged (fig. 15 (b)). Again, all pointer copies are retained, but they cease to act as pointers.

8.2. Mathematical Models

The gene assembly process described above uses a few molecular recombination operations. These operations have been formalized into three formal models [27], using the following frameworks: MDS descriptors, legal strings, and overlap graphs. All three models are equivalent, in that they describe the unscrambling of a gene and they can do so with any arbitrarily permuted gene.

1. *MDS* **descriptors**. The first formalization of gene assembly uses the macronuclear destined segments (*MDS*), as such, to formally describe the state of the gene and the sequence of operations performed to bring a gene in canonic form. *MDS*s are identified by their index, which represents their position in the canonic gene. For example, MDS_3 will be the third segment in the final gene. The segments that have no genetic meaning, that is internally eliminated segments (*IES*s), are indexed by their position in the initial micronuclear gene. The succession of segments as they appear in the gene, is called an *MDS arrangement.* In the arrangement M and I denote *MDS* and *IES* respectively. For the *actin* gene shown in fig. 10 the *MDS* arrangement is
$$M_3 I_1 M_4 I_2 M_6 I_3 M_5 I_4 M_7 I_5 M_9 I_6 \overline{M_2} I_7 M_1 I_8 M_8$$
   A composite *MDS*, $M_{i,j}$, is formed when the intermediate *MDS*s are already assembled: $M_i, M_{i+1}, ..., M_j$ are already contiguous.

   The *MDS* arrangement representation can be simplified. *MDS descriptors* keep only the delimiting pointers, as a pair (i, j). The pointers at the beginning and the end of the genes have special names b and e. For any singular *MDS* (M_i), the delimiting pointers are denoted i and $i + 1$. The following are examples of *MDS* descriptors: $M_i \rightarrow (i, i + 1)$, $M_1 \rightarrow (b, 2)$, the last *MDS*: $M_9 \rightarrow (9, e)$, and a composite $M_{i,j} \rightarrow (i, j + 1)$. A gene is completely arranged, if it is of the form (b, e) or $(\overline{e}, \overline{b})$.

   The *MDS* descriptor for actin (fig. 10) is
$$(3,4)(4,5)(6,7)(5,6)(7,8)(9,e)(\overline{3},\overline{2})(b,2)(8,9)$$
   The gene assembly operations (**ld**, **hi**, **dlad**) can actually be easily expressed with *MDS* descriptors. Generally, when an assembly operation is applied to an *MDS* descriptor, then either one pointer or two pointers are used up and disappear from the descriptor. In particular, one pointer is eliminated from the descriptor for **ld** and **hi**, and two pointers are eliminated in the case of **dlad**.

Let δ_i be an arbitrary subsequence of an MDS descriptor. Then

$$ld_p(\delta_1(q,p)(p,r)\delta_2) = \delta_1(q,r)\delta_2$$

applies a loop recombination (**ld**) on the pointer p,

$$hi_p(\delta_1(q,p)\delta_2(\overline{r},\overline{p})\delta_3) = \delta_1(q,r)\overline{\delta_2}\delta_3$$

applies a hairpin recombination (**hi**) on the pointer p, and

$$dlad_{p,q}(\delta_1(p,r_1)\delta_2(q,r_2)\delta_3(r_3,p)\delta_4(r_4,q)\delta_5) = \delta_1\delta_4(r_4,r_2)\delta_3(r_3,r_1)\delta_2\delta_5$$

applies the double loop recombination (**dlad**) on pointers p and q. Note that the 3 formulas above do not show all combinations of formulas in which the operations apply. Though all other forms are similar to the above, an exhaustive list can be found in [27].

2. **legal strings**. With legal [18] strings an even simpler model than MDS descriptors is obtained. An MDS descriptor is transformed into a legal string by dropping the parentheses (the pair construct) and the beginning and end pointers (b and e). The result is a string. Also operationally, legal strings behave like strings. For *actin*, the legal string is

$$34456756789\overline{32}289$$

Note that each legal string contains two copies of the same index. If the legal string contains a and the inverse $\overline{a}$ then a is *positive* in the string; otherwise a is *negative*. A legal string is *realistic*, if it is the equivalent of a realistic MDS descriptor (or arrangement).

Two pointers, p and q, are said to *overlap*, if exactly one copy of q appears between the two copies of p. For example, in the string

$$w = 3\overline{5}265473672\overline{4}$$

4 and 2 overlap, whereas 4 and 7 do not overlap.

The molecular operations translate into three string rules. Let u_i be a legal string.

(a) The string negative rule, snr, (equivalent to **ld**) eliminates a negative pointer p.

$$snr_p(u_1p\overline{p}u_2) = u_1u_2$$

(b) The string positive rule, spr, (equivalent to **hi**) eliminates a positive pointer p and inverts the substring between the two occurrences of the pointer.

$$spr_p(u_1pu_2\overline{p}u_3) = u_1\overline{u_2}u_3$$

(c) The string double rule, sdr, (equivalent to **dlad**) eliminates 2 pointers p and q that overlap and switches the substrings defined by the pointers' overlapping.

$$sdr_{p,q}(u_1pu_2qu_3pu_4qu_5) = u_1u_4u_3u_2u_5$$

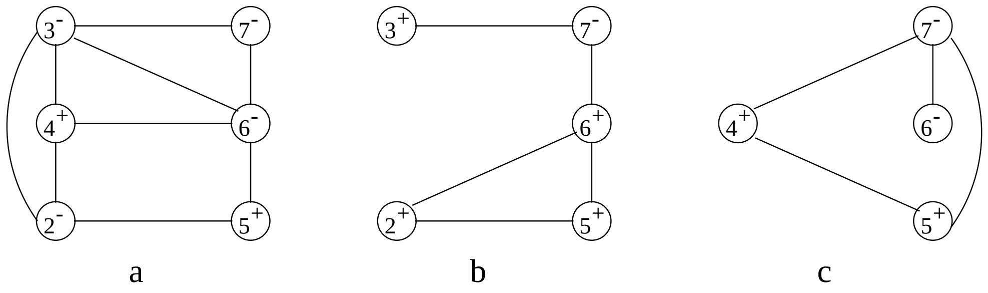

Figure 16: (a) An overlap graph. (b) After applying gpr_4 on the graph in (a). (c) After applying $gdr_{2,3}$ on the graph in (a).

3. **overlap graphs** [19]. The following abstract model of the gene assembly molecular operations can be graphically interpreted. In fact, these molecular operations are modeled as graph transformations. The graph represents the state of the scrambled gene. A ready assembled gene corresponds to the empty graph.

   There exists a direct transformation of legal strings into overlap graphs. Both use the same pointer concepts. The vertices of the graph represent pointers. Moreover, the graph is signed, in that, each vertex carries the sign + or −. A vertex p is positive (p^+), if the pointer p is positive in the legal string, and p is negative (p^-) otherwise. Two vertices p and q are connected in the graph, if the pointers p and q overlap in the legal string. The definition of the graph's edges has given the name of this abstract model, namely overlap graphs.

   Fig. 16 (a) shows the overlap graph for the following legal string:

   $$w = 3\overline{5}265473672\overline{4}$$

   For signed graphs, the molecular operations affect the number of vertices and also the set of edges. Every operation removes one or two vertices. Several edges can be affected by a single molecular operation: they are either added or removed. The following graph operations counterpart the molecular operations:

   (a) The *graph negative rule* (equivalent to **ld**) removes an isolated vertex p, provided the vertex is negative (p^-). The isolated vertex p means p does not overlap with any other pointer.

   (b) The *graph positive rule* (equivalent to **hi**) applies to a positive vertex p. The vertex is removed and all vertices in its neighborhood $N(p)$ are affected. All vertices in the neighborhood switch signs. All edges in the neighborhood are complemented: for any two vertices $q, r \in N(p)$, if they were connected by an edge, their edge is removed, whereas if they were not adjacent, a new connecting edge is added. Consider, for example, vertex 4^+ in fig. 16 (a). Vertex 4^+ has three neighbors: $N(4^+) =$

$\{2^-, 3^-, 6^-\}$. The *graph positive rule* applied on 4^+ results in the graph given in fig. 16 (b). Note that 2^+, 3^+, and 6^+ have changed signs.

(c) The *graph double rule* (equivalent to **dlad**) is applied to two adjacent vertices p and q, if p and q are both negative. The rule eliminates both vertices and affects their neighbors. All signs remain unchanged. Nevertheless, the rule changes the overlapping among vertices in the neighborhoods $N(p)$ and $N(q)$. Every edge whose ends are in $N(p)$ and $N(q)$ respectively are complemented, with the exception of an edge with both ends in $N(p) \cap N(q)$. Fig. 16 (c) shows the result of applying the *g*raph double rule to the vertices 3^- and 2^- of fig. 16 (a).

Note that *template guided gene assembly* [17] is a different mathematical model to describe the same process of gene assembly in ciliates. In this model, DNA strands are recombined under the guidance of templates. Templates are small *ss* DNAs that have to be present during gene assembly. They describe a pointer neighborhood, where a recombination is to take place. It is not clear, from experiments, which model, *template guided recombination* or *pointer based recombination*, better describes reality.

9. Conclusion

This work has presented Biomolecular Computing in terms of theoretical studies and practical results. All three branches, namely, DNA computing, membrane computing and gene assembly in ciliates, rely on strong, well-defined mathematical models. They all promise the capacity of computing everything a Turing machine does. In terms of practical realisation of a computing device, DNA computing engages the largest efforts to bring forth a viable, either large or limited applicability, computer. Membrane computing can recommend itself only with simulations on standard computers; implementations of P systems in living cells seem to be far from realisation. In terms of practical implementation, the situation of gene assembly in ciliates is the most interesting, as the system is *already* implemented naturally in unicellular organisms. The process of gene assembly is not controllable by humans though. The table below shows a summary of the comparison of three Biomolecular Computing branches.

	Mathematical model	*Practical implementations*	*Human designed*
DNA computing	Yes	Yes	Yes
Membrane computing	Yes	No	Yes
Ciliates	Yes	Yes	No

The long term goal of DNA computing is to have DNA computers with wider applicability. All experiments today are small procedures with small size input data.

To make the transition towards reasonable size applications means to solve several technical open issues:

1. Although many molecular manipulation techniques are standard, they have less than 100 % accuracy. These errors tend to accumulate and become significant for longer programs. There is a need of even a small set of *automated, accurate* molecular manipulation techniques.

2. *Scalability* is an issue in all implementations so far. The exhaustive search procedures use exponential space for a give input size. Therefore, implementations are very sensitive to even small increases in the input. Managing larger implementations of already existing procedures is challenging.

3. All search algorithms implemented generate the input data probabilistically. The solution strand is not guaranteed to be created in the initial data pool. A new technique that would generate all candidate solutions *deterministically* would be welcome.

4. Finally, most algorithms, researched theoretically and also implemented, are search algorithms. It is not clear, whether DNA computing could efficiently address other types of computation. In particular, it is an open question what super-Turing computation problems can be answered by DNA algorithms.

References

[1] Web page of P systems: http://psystems.disco.unimib.it.

[2] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[3] Leonard Adleman. Computing with DNA. *Scientific American*, pages 34–41, August 1998.

[4] Selim G. Akl. From infinitely small to infinitely big: The universe as computer. Public lecture, Queen's University, November 2005.

[5] Selim G. Akl. The myth of universal computation. In R. Trobec, P. Zinterhof, M. Vajteršic, and A. Uhl, editors, *Parallel Numerics, Part 2, Systems and Simulation*, pages 211–236. University of Salzburg, Austria and Jožef Stefan Institute, Ljubljana, Slovenia, 2005.

[6] Selim G. Akl. Inherently parallel geometric computations. *Parallel Processing Letters*, 16(1):19–37, March 2006.

[7] Selim G. Akl, Brendan Cordy, and W. Yao. An analysis of the effect of parallelism in the control of dynamical systems. *International Journal of Parallel, Emergent and Distributed Systems*, 20(2):147–168, June 2005.

[8] Donald Beaver. Computing with DNA. *Journal of Computational Biology*, 2:1–7, 1995.

[9] F. Bernardini and M. Gheorghe. Population P systems. *Journal of Universal Computer Science*, 10(5):509–539, 2004.

[10] Mireille Bétermier. Large-scale genome remodelling by the developmentally programmed elimination of germ line sequences in the ciliate paramecium. *Research in Microbiology*, 155:399–408, 2004.

[11] Dan Boneh, Christopher Dunworth, Richard J. Lipton, and Jiří Sgall. Making DNA computers error resistant. Department of Computer Science, Princeton University, Princeton, USA, and Mathematical Institute, AV ČR, Praha, Czech Republic.

[12] Dan Boneh, Cristopher Dunworth, and Richard J. Lipton. Breaking DES using a molecular computer. Technical Report 489-95, Princeton CS, Princeton, NJ, 1995.

[13] Ravinderjit S. Braich, Nickolas Chelyapov, Cliff Johnson, Paul W.K. Rothemund, and Leonard Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296:499–502, 2002.

[14] M. Cavaliere. Evolution-communication P systems. In Gh. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing. International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania*, pages 134–145. Springer-Verlag, 2003.

[15] Weng-Long Chang, Minyi Guo, and Jesse Wu. Solving the independent-set problem in a DNA-based supercomputer model. *Parallel Processing Letters*, 15(4):469–479, 2005.

[16] L. Colson, N. Jonoska, and M. Margenstern. λ P systems and typed λ -calculus. In G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing. International Workshop, WMC5, Milan, Italy, 2004. Revised Papers, Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.

[17] Mark Daley and Ian McQuillan. Template-guided DNA recombination. *Theoretical Computer Science*, 330(2):237–250, 2005.

[18] A. Ehrenfeucht, Tero Harju, Ion Petre, D.M. Prescott, and Grzegorz Rozenberg. *Computation in Living Cells: Gene Assembly in Ciliates.* Springer, 2003.

[19] A. Ehrenfeucht, Tero Harju, Ion Petre, D.M. Prescott, and Grzegorz Rozenberg. Formal systems for gene assembly in ciliates. *Theoretical Computer Science*, 292, 2003.

[20] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[21] Dirk Faulhammer, Anthony R. Cukras, Richard J. Lipton, and Laura F. Landweber. Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences of the United States of America*, 97(4):1385–1389, February 2000.

[22] R. Freund and A. Păun. Membrane systems with symport/antiport rules: Universality results. In Gh. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing. International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania*, pages 270–287. Springer-Verlag, 2003.

[23] Frank Guarnieri, Makiko Fliss, and Carter Bancroft. Making DNA add. *Science*, 273(12):220–223, July 1996.

[24] M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, and A. Riscos-Núñez. On descriptive complexity in P systems. In *Preproceedings of the 5-th Workshop on Membrane Computing, Milano-Italy (WMC5)*, pages 245–255, June 2004.

[25] M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, and J. Romero-Campero. A linear solution for QSAT with membrane creation. In *Preproceedings of the 6-th Workshop on Membrane Computing, Vienna-Austria (WMC6)*, pages 395–409, July 2005.

[26] Masami Hagiya. From molecular computing to molecular programming. In *Proceedings of the 6-th DIMACS Workshop on DNA Based Computers, held at University of Leiden*, pages 87–100, 2000.

[27] Tero Harju, Ion Petre, and Grzegorz Rozenberg. Gene assembly in ciliates: Formal framework. TUCS Technical Report 558, Turku Centre for Computer Science, October 2003.

[28] Tero Harju, Ion Petre, and Grzegorz Rozenberg. Gene assembly in ciliates: Molecular operations. TUCS Technical Report 557, Turku Centre for Computer Science, October

2003.

[29] Tom Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49:737–759, 1987.

[30] Peter Kaplan, Guillermo Cecchi, and Albert Libchaber. Molecular computation: Adleman's experiment repeated. Technical Report 95-120, NEC Research Institute, Princeton, NJ, 1995.

[31] Lila Kari, Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and S. Yu. DNA computing, sticker systems, and universality. *Acta Informatica*, 35(5):401–420, 1998.

[32] Shankara N. Krishna and Raghavan Rama. P systems with replicated rewriting. *Journal of Automata, Languages and Combinatorics*, 6(3):345–350, 2001.

[33] Stuart A. Kurtz, Stephen R. Mahaney, James S. Royer, and Janos Simon. Biological computing. In *Complexity Theory Retrospective II*, pages 179–196. Springer-Verlag, 1997.

[34] Benjamin Lewin. *Genes VII.* Oxford University Press, 2000.

[35] Richard J. Lipton. DNA solution of hard computational problems. *Science*, 268(5210):542–545, April 1995.

[36] Qinghua Liu, Liman Wang, Anthony G. Frutos, Anne E. Condon, Robert M. Corn, and Lloyd M. Smith. DNA computing on surfaces. *Nature*, 403:175–179, January 2000.

[37] Qi Ouyang, Peter D. Kaplan, Shumao Liu, and Albert Libchaber. DNA solution of the maximal clique problem. *Science*, 278:446–449, October 1997.

[38] M.J. Pérez-Jiménez and A. Riscos-Núñez. A linear solution for the knapsack problem using active membranes. In *Membrane Computing. Lecture Notes in Computer Science*, volume 2933, pages 250–268. 2004.

[39] D. M. Prescott, A. Ehrenfeucht, and Grzegorz Rozenberg. Molecular operations for DNA processing in hypotrichous ciliates. *European Journal of Protistology*, 37, 2001.

[40] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences.*

[41] Gheorghe Păun. On the splicing operation. *Discrete Applied Mathematics*, 70:57–79, 1996.

[42] Gheorghe Păun. Regular extended H systems are computationally universal. *Journal of Automata, Languages, Combinatorics*, 1(1):27–36, 1996.

[43] Gheorghe Păun. P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages, Combinatorics*, 6(1):5–90, 2001.

[44] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing - New Computing Paradigms.* Springer, 1998.

[45] John H. Reif. Perspectives: Computing. successes and challenges. *Science*, 296, Apr 2002.

[46] Grzegorz Rozenberg. DNA processing in ciliates - the wonders of DNA computing *in vivo.* 1. Leiden Center for Natural Computing, Leiden University, The Netherlands, 2. Departement of Computer Science, university of Colorado, Boulder, USA.

[47] Kensaku Sakamoto, Hidetaka Gouzu, Ken Komiya, Daisuke Kiga, Shigeyuki Yokoyama, Takashi Yokomori, and Masami Hagiya. Molecular computation by DNA hairpin formation. *Science*, 288(19):1223–1226, May 2000.

[48] Shetu N. Shah and Michael T. Niemier. Pattern matching with DNA computers - draft. College of Computing, Georgia Institue of Technology, Atlanta, Georgia, USA.

[49] Claude Shannon. *Bell Syst. Tech. J.*, 28(59), 1949.

[50] Kenneth D. Stuart, Achim Schnaufer, Nancy Lewis Ernst, and Aswini K. Panigrahi. Complex management: RNA editing in trypanosomes. *Trends in Biochemical Sciences*, 30(2):97–105, February 2005.

[51] Lionel Tarassenko. *A Guide to Neural Computing Applications.* Elsevier, 1998.

[52] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory.* MIT Press, 1999.

[53] Hiroshi Yoshida and Akira Suyama. Solution to 3-SAT by breadth first search. In Erik Winfree and David K. Gifford, editors, *Proceedings of the 5-th DIMACS Meeting on DNA Based Computers*, MIT, June 1999.

[54] Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri. Solving NP-complete problems using P systems with active membranes. In I. Antoniou, C.S. Calude, and M.J. Dinneen, editors, *Unconventional Models of Computation*, pages 289–301. Springer, London, 2000. DISCO - Universita di Milano-Bicocca, Italy.