2D DNA Self-Assembly for Satisfiability

Michail G. Lagoudakis and Thomas H. LaBean
Department of Computer Science
Duke University
Durham, NC 27708
{mgl,thl}@cs.duke.edu

Abstract

DNA self-assembly has been proposed as a way to cope with huge combinatorial NP-HARD problems, such as satisfiability. However, the algorithmic designs for DNA self-assembly proposed so far are highly dependent on the instance to be solved. The required work (DNA synthesis, tile construction, encoding, etc.) can be done only after the instance is given. This paper presents an algorithmic design for solving satisfiability problems using two-dimensional DNA self-assembly (tiling). The main driving factor in this work was the design and encoding of the algorithm in a general way that minimizes the dependency on particular instances. In effect, a large amount of work and preparation can be done in advance as a batch process in the absence of any particular instance. In practice, it is likely that the total time (from the time an instance is given, to the time a solution is returned) will be decreased significantly and laboratory procedures will be simplified.

1 The Satisfiability (SAT) Problem

The *Boolean Satisfiability* (SAT) problem is the most known representative of the NP-HARD class of problems. The non-polynomial (usually exponential) time required for optimal solutions to these problems, implies that solution of large instances becomes intractably difficult, if not practically impossible.

A SAT instance consists of a number of Boolean variables $x_1, x_2, ..., x_m$ and a number of clauses $C_1, C_2, ..., C_n$. Each clause is a disjunction of distinct literals, whereby a literal is a single variable x_i itself or its complement $\bar{x}_i$. A solution (satisfying assignment) is an assignment of binary values to the variables x_i, such that the conjunction of all clauses is satisfied. Boolean formulas represented in this format are said to be in *Conjunction Normal Form* (CNF); a conjunction of disjunctions.

An example with 5 variables and 8 clauses is given below:

$$(x_1 + x_2 + \bar{x}_3)(\bar{x}_2 + x_4)(\bar{x}_1 + \bar{x}_5)(\bar{x}_1 + x_2 + x_3 + \bar{x}_4 + x_5)(\bar{x}_3)(x_2 + x_5)(\bar{x}_5)(x_1)$$

In this case, the assignment $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 1, 0)$ is satisfying.

So far, there is no restriction on the number of literals that a single clause can contain. However, even if we require that each clause has exactly k literals ($k \geq 3$), the problem (known as kCNF-SAT) is still intractable. Moreover, it is proven that any SAT instance can be turned into an equivalent kCNF-SAT instance [GaJo79]. From these variations, most notable is the 3CNF-SAT variation, because of the small-sized clauses. Notice that for a given number of m variables, there can be

$$\binom{m}{k} \times 2^k$$

possible clauses in a kCNF-SAT formula (choose k variables and for each one either leave intact or negate). In particular, for $k = 3$, there are

$$\binom{m}{3} \times 2^3 = \frac{4}{3} \times m(m-1)(m-2) = O(m^3)$$

possible clauses in a 3CNF-SAT formula.

2 DNA Computation and Satisfiability

The basic idea is to exploit the massive parallelism present in DNA operations in order to emulate a non-deterministic device that solves the SAT problem in polynomial time. Consider a particular assignment to the boolean variables in the formula. On a conventional computer it is fairly easy to check whether this particular assignment is a solution to the problem, i.e. an assignment that satisfies all the clauses in the formula. In fact, this can be done in time linear to the size of the formula. It is the huge (exponential) number of different assignments that makes the problem difficult.

This work proposes a way to perform this checking procedure on molecular substrate using 2D DNA self-assembly. By creating billions of copies of the participating DNA structures (tiles, in our case), we expect that this procedure will run in parallel on all possible assignments. The assignments will be created dynamically as part of the assembly. In effect, that will make the computation time linear to the size of the formula, while pushing the exponential dimension of the problem into the large number of DNA assemblies, and thus into the space (volume) occupied by the DNA molecules. If there is a satisfying assignment, we expect that at least one of these parallel checks will discover it.

This section reviews the main proposals for biomolecular solutions to the SAT problem, briefly describes the general DNA structures and methods that are used here and delineates our work.

2.1 Related Work

Lipton was the first to propose a DNA model for satisfiability. His proposal [Lipt95] is based on Adleman's elimination method, whereby the whole combinatorial space of solutions is created and subsequently the "good" ones are extracted by a series of separation steps. Later, Hagiya et al. [Hagi97] presented an approach to evaluate and learn μ-formulas (a particular form of Boolean formulas) using a technique that is commonly known as Whiplash PCR. This method was improved by Winfree [Winf98]. Finally, a proposal for CNF-SAT using hairpin DNA tiles and linear assembly can be found in [WiER98].

A common feature of these approaches and a potential practical problem is that construction of the participating DNA structures cannot really begin until the particular problem instance at hand is given. This "instance-specific" design implies large total computation time (encoding/DNA computation/decoding). In addition, all required man/machine resources need to be employed again and again as new instances are provided. Moreover, the specificity of the encoding step will increase the likelihood of encoding errors, which if occur will render the whole computation useless.

A possible solution to these problems, would be an algorithmic design that requires minimal encoding for a given instance (some straightforward description of the input), whereas the main algorithm is coded in preconstructed "instance-independent" DNA molecules that can be created and even tested off-line in a batch fashion.

The DNA self-assembly technique we are going to use is known as DNA tiling computation and was proposed by Eric Winfree [Winf98]. Basic components for DNA tiling have been prototyped and tested by Nadrian Seeman and his colleagues. In particular, double crossover molecules have been shown to be rigid and able to form planar lattices [WLWS98]. Winfree showed how to solve the Hamiltonian Path problem using 2-dimensional DNA tiling [Winf98]. Further, LaBean, Winfree and Reif [LaWR99] have been experimenting with parallel XOR and addition operations using DNA tiling.

2.2 2D DNA Self-Assembly for Satisfiability

The main goal of this algorithmic design is to avoid the high degree of specificity that characterizes the DNA structures of the previous approaches. This could be accomplished by coding the general algorithm as a library of non-specific DNA tiles which when combined with the appropriate encoding of the given instance (input) would perform the desired computation. A separation step afterwards could separate the successful computations from where a satisfying assignment could be drawn. Put it another way, we would like an encoding that codes the general algorithm for the problem and not one that codes a specialized algorithm that solves a particular instance of the problem.

Our design is described at the algorithmic level. We abstract each DNA tile as a square with labels at the corners (see figure below). Each label indicates a particular kind of a sticky end. Two sticky ends that can match and ligate correspond to identical labels. Each tile can have any from 1 to 4 labels. Non-labeled corners indicate non-sticky ends. Experimental work [Winf98] has shown that, in principle, there exist parameter (temperature) conditions under which assembly in a slot is favored when both participating sticky ends match. Although these conditions are still difficult to achieve in practice, this result shows that undesired and/or corrupted assemblies can be avoided. It is taken for granted here that a tile would not occupy a slot unless both labels match.

Figure 1: Abstracted Tile Representation.

The actual implementation details are not discussed here since they fall outside of the scope of this manuscript. Eventually, we would like to test our claims experimentally. However, we believe that we make no arbitrary hypotheses. In fact, our work is based on the assumptions and achievements of Winfree, Seeman, LaBean and Reif. The validity of their approaches will logically imply validity of ours.

2.3 The Non-Deterministic Algorithm

As it was mentioned our design attempts to implement a non-deterministic algorithm for the SAT problem. Non-determinism implies that at some step(s) the algorithm makes a non-deterministic choice (as if some oracle could tell you what to choose). The algorithm is given below. Notice that step 3 is the nondeterministic step.

NonDeterministic Satisfiability($x_1, x_2, \ldots, x_m, C_1, C_2, \ldots, C_n$)

1. Set all clauses to be unmarked
2. **for** (each boolean variable $x_i, i = 1, \ldots, m$) **do**
3. Assign a value (0 or 1) to variable x_i
4. Mark all clauses $C_j, j = 1, \ldots, n$ which are satisfied by this assignment
5. **if** (all clauses are marked)
6. **then return** YES
7. **else return** NO

We present two slightly different designs for encoding this algorithm as DNA tiling. The difference is on what the tiles code. Design A is based on an encoding of clauses, whereas Design B is focusing on literal encoding. After a detailed exposition of both designs a comparison follows.

3 Design A: Encoding Clauses

In this case we assume that the formula is given in 3-CNF form[1]. Since the number of possible clauses is countable in this case, we can order all clauses and number them in some systematic way. With this mapping each clause C is represented by its corresponding number, say j. For simplicity, in what follows we number a clause C with its number in the ordering as C_j. Given a variable x_i and a clause C_j, we can easily construct a function $F(i, v, j)$ that determines whether the clause C_j is satisfied when variable x_i takes the binary value v. The function F will be actually "precoded" in the structure of the DNA tiles.

We can represented the desired computation in a table format, that facilitates the transition to the DNA assembly. Consider the 3-CNF formula

$$(\bar{x}_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2 + x_3)$$

and the table given below. There are 3 variables (represented in the first column of the table) and 3 clauses (represented in the last row) with numbers 1, 4, and 3 according to some ordering. The second column represents a possible assignment. Cells marked with "*" are helper cells.

*	T	T	T	T	SS
$\mathbf{x_3}$	1	OK	OK	OK	*
$\mathbf{x_2}$	0	OK	OK	C_3	*
$\mathbf{x_1}$	1	C_1	OK	C_3	*
*	*	$\mathbf{C_1}$	$\mathbf{C_4}$	$\mathbf{C_3}$	*

Following the algorithm above, the table is filled in a bottom-up manner, one row at a time. A cell corresponding to variable x_i and clause C_j will be marked as "OK" if and only if the clause C_j is satisfied when variable x_i takes the binary value v_i indicated in the second column or the cell below is already labeled "OK". Otherwise, it is marked with the clause name "C_j". Therefore, as we move up, "C_j" is propagated up as long as the clause is not satisfied. Once it is satisfied, it turns to an "OK" label, which propagates to the top independently of the assignment to the remaining variables. In effect, the entries of the table reflect the function F.

It remains to check whether all clauses are satisfied. This is done in the first row of the table. Initially, we assume that the conjunction of all clauses is satisfied (label "T"=TRUE in the second column)[2]. The label "T" will propagate to the right as long as there are "OK"s to the right in the row below. The upper right cell is filled with "SS" (=SUCCESS) if and only if the cell to the left is "T" and the cell below is "*" (end of formula). Therefore, the formula is satisfied with this assignment if and only if the symbol "SS" appears in the table. Notice that if it was not satisfied, "T" would not had propagated to the end and "SS" would never appear, in which case we could propagate a label "F" (=FALSE) or leave the table incomplete.

The idea illustrated with the table above can be carried out with 2D assembly of DNA tiles, provided the appropriate tiles. The input is coded as a concatenation of tiles representing the first column and the last row of the table. This input structure is reproduced in billions and is mixed with a DNA solution that already contains tiles from a fixed library to be described shortly. The appropriate tiles will self-assemble on this input layer. Values are assigned to the variables in a random manner. Each assembly is testing one possible assignment. The input DNA structure and the resulting tiling computation of a satisfying assignment (in fact, the one in the table above) is shown in Figure 2.

For illustration purposes, the whole computation is unfolded step-by-step in Figure 3. Initially, there is only one slot where a tile can bind. This tile will be an assignment time; the label "V" (=Value) indicates that a value is expected for the particular variable. Once a value is assigned to x_1,

[1] Generalization to kCNF-SAT can be easily done, albeit increasing the number of required DNA tiles.

[2] By definition, an empty conjunction is satisfied [DaSW83].

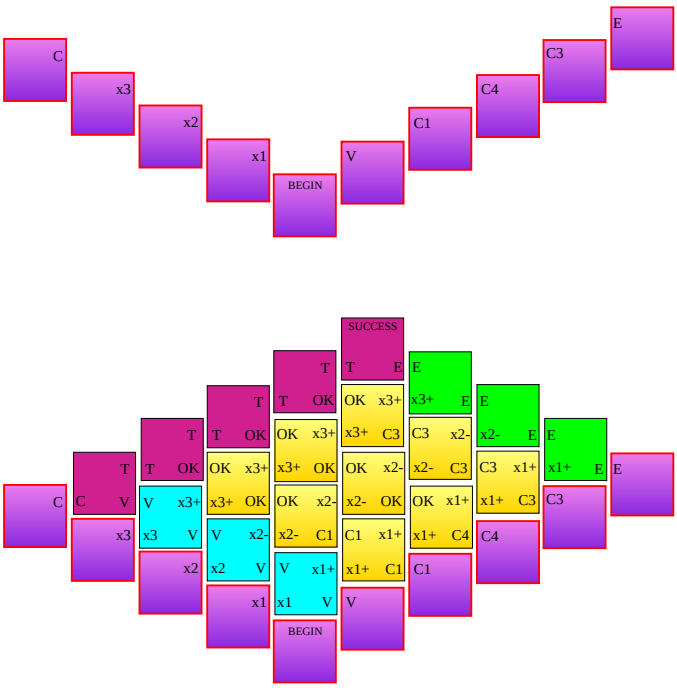


Figure 2: Input tiles and a successful assembly computation for design A.

there are two open slots. The slot to the left "asks" for an assignment to variable x_2. The slot to the right "wants" to check whether the assingment of 1 to x_1 (shown as x_1+) satisfies clause C_1. This slot will be filled by the appropriate tile that contains the "answer". Unfortunately, x_1+ does not satisfy C_1 and thus the label C_1 is propagated as is to be checked against the remaining assignments. At the same time, x_1+ is propagated at the other side to check for the remaining clauses. At the third step, there are three slots open. One for assigning a value to x_3, one for checking C_1 against x_2-, and one for checking C_4 against x_1+. Notice that both clauses are satisfied in this case, and thus "OK" is propagated up and left. This continues until assignment of values has been completed at which time the final check begins as well (indicated by label "C" for Check). If all clauses are satisfied the assembly will continue to the "T" (=True) label will meet the "E" (=End) label and the success marker will be placed on the top. If there is some unsatisfied clause, the "T" label cannot propagate and the assembly will remain incomplete and thus without the success marker. At the very end, a separation procedure that isolates the assemblies that contain the success marker will provide a solution(s) to the input instance.

3.1 Complexity of Design A

The complexity of the design is considered in terms of computation time, computation space and number of distinct tiles required. It is obvious from the examples given that the computation time $T(A)$ is equal to the depth (diagonal) of the assembly. In fact, it is

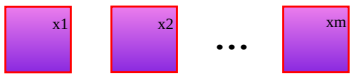
$$T(A) = (m+1) + (n+2) - 1 = m + n + 2 = \Theta(m+n) = O(m^3)$$

for m variables and n clauses. $\Theta(m+n)$ is linear to the size of the formula. $O(m^3)$ is an upper bound polynomial to the number of variables. We have used the fact that that $n = O(m^3)$ for 3CNF-SAT (see section 1). The space $S(A)$ taken for each assembly is the area of the assembly.

$$S(A) = (m+2) \times (n+3) = \Theta(m \times n) = O(m^4)$$

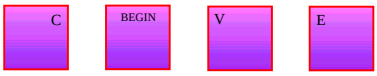
which is upper-bounded polynomially to the number of variables. Finally, the library of fixed titles need contain the following tiles:

- **Variables.** There have to be m tiles coding m variables, where m is the maximum number of variables that can appear in a formula.

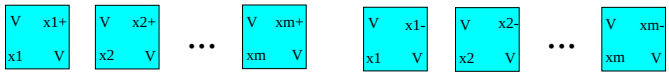


- **Clauses.** There has to be $n = \frac{4}{3} \times m(m-1)(m-2) = \Theta(m^3)$ tiles coding all possible clauses.

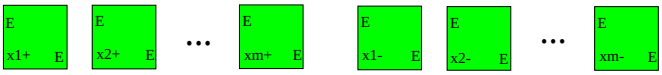
- **Input Boundaries.** There are 4 such tiles to mark the end of variable list, the end of clause list, the beginning of value assignment and the beginning of computation in the input assembly.



- **Assignment.** Each variable can take one of two values, so there is a total of $2m$ tiles for assigning values. A $+$ sign indicates an assignment of 1, whereas a $-$ sign indicates an assignment of 0.



- **Computation Boundaries.** For each variable assignment there has to be an ending tile, thus a total of $2m$ tiles.



- **Computation.** For each variable assignment and for each clause there has to be a tile that indicates whether the clause is satisfied or not. There are $2mn$ such tiles. Further, there has to be tiles to propagate the "OK"s of the satisfied clauses to the end of the assembly. There are $2m$ of those tiles.

- **Final Check.** Finally, there have to be some tiles to check if all clauses are satisfied and mark the result. There are 3 of those shown below.

Summing up all the numbers, we have a total number of tiles:

$$N(A) = 7m + (2m+1)n + 7 = \Theta(m \times n) = \Theta(m^4)$$

3.2 Example

In order to reinforce the method, Figure 4 shows another example of successful computation for a formula with 8 variables and the following 10 clauses.

$$
\begin{aligned}
C_1 &= (\bar{x}_2 + x_3 + x_7) \\
C_2 &= (x_3 + \bar{x}_5 + \bar{x}_8) \\
C_3 &= (\bar{x}_1 + x_2 + x_4) \\
C_4 &= (\bar{x}_3 + x_5 + x_6) \\
C_5 &= (x_1 + \bar{x}_4 + x_5) \\
C_6 &= (x_5 + x_7 + x_8) \\
C_7 &= (\bar{x}_2 + x_4 + x_6) \\
C_8 &= (x_3 + \bar{x}_4 + x_5) \\
C_9 &= (x_1 + x_4 + \bar{x}_6) \\
C_{10} &= (x_2 + \bar{x}_6 + x_8)
\end{aligned}
$$

4 Design B: Encoding Literals

This design attempts to overcome the large number of tiles required by design A. In particular, it operates at a lower level encoding literals that appear in each clause rather than the clause itself. The basic idea runs the same way. Consider the same 3-CNF formula

$$(\bar{x}_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2 + x_3)$$

and the table given below. There are 3 variables (represented in the first column of the table) and 3 clauses (represented in the last row) with all literals listed explicitly. The "s" (=SEPARATOR) is used to separate clauses. The second column represents a possible assignment as before. Cells marked with "*" are helper cells.

*	F	F	T	T	F	T	T	T	F	F	F	T	F	SS
$\mathbf{x_3}$	1	$\bar{x}_1$	OK	OK	s	OK	OK	$\bar{x}_3$	s	$\bar{x}_1$	x_2	OK	s	*
$\mathbf{x_2}$	0	$\bar{x}_1$	OK	x_3	s	OK	OK	$\bar{x}_3$	s	$\bar{x}_1$	x_2	x_3	s	*
$\mathbf{x_1}$	1	$\bar{x}_1$	$\bar{x}_2$	x_3	s	OK	$\bar{x}_2$	$\bar{x}_3$	s	$\bar{x}_1$	x_2	x_3	s	*
*	*	$\mathbf{\bar{x}_1}$	$\mathbf{\bar{x}_2}$	$\mathbf{x_3}$	s	$\mathbf{x_1}$	$\mathbf{\bar{x}_2}$	$\mathbf{\bar{x}_3}$	s	$\mathbf{\bar{x}_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	s	*

Again, following the nondeterministic algorithm, the table is filled in a bottom-up manner, one row at a time. However, now a cell corresponding to variable x_i and literal y_j will be marked with "OK" if and only if literal y_j is TRUE when variable x_i takes the binary value v_i (indicated in the second column) or the cell below is already labeled "OK". Otherwise, it is marked with the literal name "y_j" to propagate the computation to the next row. At the end of the day, the second row of the table will contain "OK"s and/or some "y_j"s depending on what satisfies what.

The checking step is a little more involved compared to the previous one. We need to check whether each individual clause is satisfied and further whether the whole formula is satisfied. Since each clause is a disjunction, we initially assume that it is not satisfied[3]. This is denoted by the "F" (=FALSE) label in the second column. As long as the particular clause has not been satisfied (i.e., there are y_j's to the right and bottom of "F"), the label "F" propagates to the right unchanged. Once,

[3]By definition, an empty disjunction is not satisfied [DaSW83]

an "OK" label is encountered to the right and bottom of an "F", it turns to a "T" (=TRUE) label, indicating that the clause has been satisfied. "T" propagates until the separator symbol "s" is met to the bottom-right. If "T" meets the separator, that implies that the current clause is satisfied and we can continue with the next one by initializing to "F" again. However, if "F" meets the separator, that means that this assignment failed to satisfy the formula and computation is halted. Finally, if the "F" label hits the "*" marker at the end of the table, it is implied that the whole formula is satisfied and therefore the success symbol "SS" marks the upper-right corner of the table. As previously, an assignment is satisfying if and only if the symbol "SS" appears at the upper-right corner of the table.

Having the concept of the table in mind, it is easy to make the transition to the DNA assembly. The basic idea is again the same as previously, but the tiles and the coding are somewhat different. Figure 5 below shows the input tile assembly and the successful computation of the table above.

Notice that by encoding literals we are not restricted to 3CNF-SAT (or to any kCNF-SAT) anymore. We can encode any SAT formula given in CNF format in a straightforward manner.

4.1 Complexity of Design B

Again, the complexity of the algorithm is considered in terms of computation time, computation space and number of distinct tiles required. The computation time $T(B)$ is

$$T(B) = (m+1) + (l+n+2) - 1 = m + l + n + 2 = \Theta(m+l+n) = O(m+mn+n) = O(mn)$$

The last equality follows from the fact that in the worst case all clauses contain all the variables, that is $l = mn$, and therefore mn is an upper bound for l. If we consider 3CNF-SAT only, then $l = 3n$ and therefore

$$T(B) = (m+1) + (4n+2) = \Theta(m+n) = O(m^3)$$

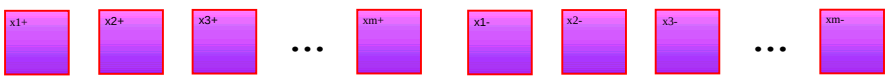
which is of the same order as in the previous design. The space $S(A)$ taken for each assembly is

$$S(B) = (m+2) \times (l+n+3) = \Theta(m(l+n)) = O(m^2 n)$$

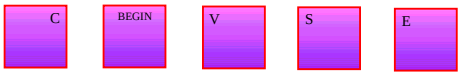
For 3CNF-SAT, $l = 3n$ and so $S(B) = \Theta(mn) = O(m^4)$ as before. The tile library is somewhat different in this case.

- **Variables.** There are m tiles coding the m variables, where m is the maximum number of variables that can appear in a formula.

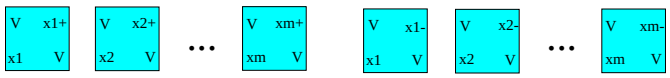
- **Literals.** There are $2m$ tiles coding all possible literals. There are used for coding the clauses of the formula.



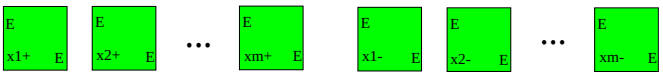
- **Input Boundaries.** There are 5 such tiles: four are the same as in design A, plus the separator.



- **Assignment.** The same as before. There is a total of $2m$ such tiles.



- **Computation Boundaries.** As before, for each variable assignment there has to be an ending tile, thus a total of $2m$ tiles.



- **Computation.** For each pair of literals x_i, y_j there has to be a single tile. If the literals do not refer to the same variable, or they refer to the same variable but are complementary, the result is simply propagation of the labels along the diagonals of the tile. However, if they match, x_i is propagated and the literal y_j is turned to an "OK". These tiles are

$$\binom{2m}{2} = m(2m-1)$$

  Further, for each variable and each possible assignment there have to be tiles to propagate the "OK" labels up. There are $2m$ of those tiles.

- **Final Check.** Finally, the tiles that check for satisfiability at the end are a little more involved than in the previous case. First, "F" has to propagate over literals y_j and turn into an "T" if "OK" is encountered. "T" propagates over y_j's and "OK"s. Finally, two tiles are needed for the extrema and one for resetting "T" to "F". In total, $4m + 5$ tiles.

Summing up all the numbers, we have a total number of tiles:

$$N(B) = 2m^2 + 12m + 10 = \Theta(m^2)$$

which is two orders of magnitude less than the total number of tiles in the previous design.

5 Comparison and Discussion

The two designs implement the same non-deterministic algorithmic in a slightly different way. It is our belief that design B is better compared to design A for two main reasons: (1) the input formula can be any CNF formula, and (2) the total number of required tiles is only $\Theta(m^2)$, where m is the maximum number of variables. In contrast, design A assumes that the formula is given in 3CNF and requires $\Theta(m^4)$ tiles. On the other hand, design A results in smaller computation time and space but asymptotically it seems that the difference disappears (see the analysis above).

There is a final detail that is crucial for the success of both algorithms. The concentrations of assignment tiles corresponding to variable x_i, i.e. tiles for x_i+ and x_i-, have to be equal so that there is equal chance of assigning either value. If this is not the case, there might be assignments that will never be explored because of this "discrimination" in assigning values.

A limitation of the algorithm, which is common for most DNA computations, comes from the fact that the exponential dimension of the problem has been pushed into the physical space (volume)

occupied by the DNA molecules. This will eventually become a restrictive factor. The input size and thus the DNA volume cannot grow forever. This implies an upper bound to the size of instances that can be solved in practice. Obviously, the practicality of a DNA algorithm for satisfiability is heavily dependent on whether this upper bound is well above the upper bound for instances that can be solved on a conventional computer.

6 Future Work

Our ultimate goal is to test the design(s) experimentally. Encouragement comes from the recent investigations of several DNA tile structures. In particular, TAO35 (see figure 1) is a general DNA tile that is currently being investigated for use in self-assembly computations [LaWR99]. Moreover, it was recently demonstrated by LaBean, Winfree and Reif [LaWi99] that input layers like the ones we use can be constructed relatively easy using as a backbone a long DNA scaffold strand that traverses all input tiles; input layer tiles assemble around this scaffold strand.

Another line of research will focus on ways to enhance the algorithm will well-known heuristics for satisfiability, such as the *unit propagation* rule (if there is a clause with a single literal, force the corresponding variable to take the value that makes the clause true) and the *purification* rule (if a variable appears in the formula in exclusively negated or non-negated form, assign to this variable the value that makes all instantiations true). Actually, both of these rules can be taken into account in the current designs simply by altering the concentrations of the assignment tiles to the corresponding variables so that only the desired value is given as an option. However, this way it becomes a manual step that is perform only at the beginning. Alternatively, preprocessing of the formula could eliminate such variables. Our goal is to incorporate them in the algorithm since the need for unit propagation and/or purification might reappear during computation.

Acknowledgements

Michail G. Lagoudakis was partially supported by the Lilian-Boudouri Foundation in Greece. This work was also supported in part by grant NSF/DARPA CCR-9725021.

References

[DaSW83] Davis, M., Sigal, R. and Weyuker, E. *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science*, Academic Press, 1983.

[GaJo79] Garey, M. and Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[Hagi97] Hagiya, M. et al. "Toward Parallel Evaluation and Learning of Boolean μ-formulas with Molecules", *Proceedings of the 3rd DIMACS meeting on DNA-based Computers*, University of Pennsylvania, 1997.

[LaWi99] LaBean, T., Winfree, E. and Reif, J. Experimental Progress in Computation by Self-Assembly of DNA Tilings, submitted to the *5th DIMACS meeting on DNA-based Computers*, MIT, 1999.

[Lipt95] Lipton, R. "DNA Solution of Hard Combinatorial Problems", *Science*, **268** (5210), 1995, pp. 542-545.

[WiER98] Winfree, E., Eng, T. and Rozenberg, G. "String Tile Models for DNA Computing by Self-Assembly", unpublished manuscript, 1998.

[Winf98] Winfree, E. *Algorithmic Self-Assembly of DNA*, Ph.D. dissertation, California Institute of Technology, 1998.

[WLWS98] Winfree, E., Liu, F., Wenzler, L. and Seeman, N. "Design and Self-Assembly of Two-Dimensional DNA Crystals," *Nature*, **394**, 1998, pp. 539-544.

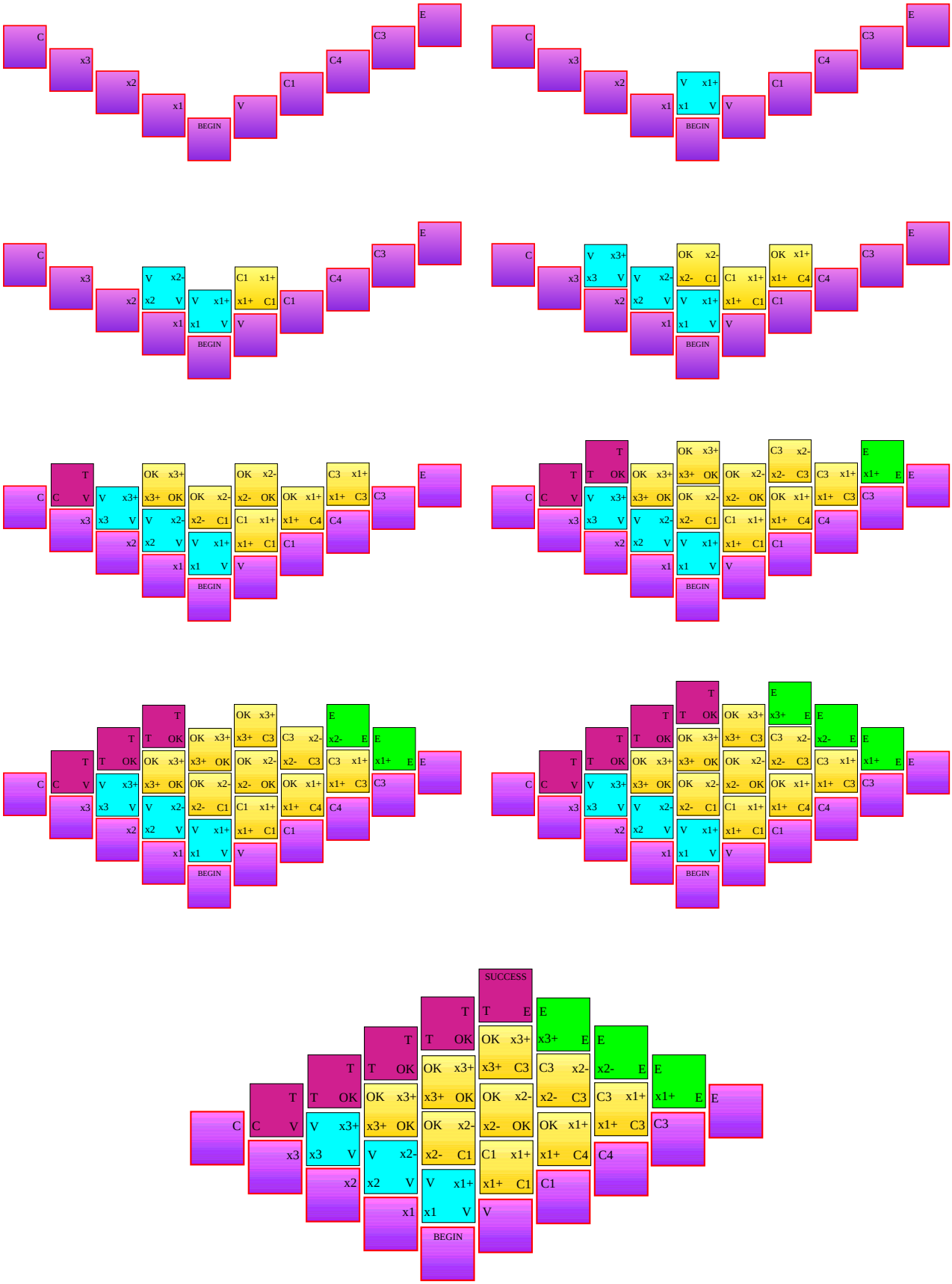


Figure 3: Steps of a successful DNA assembly.

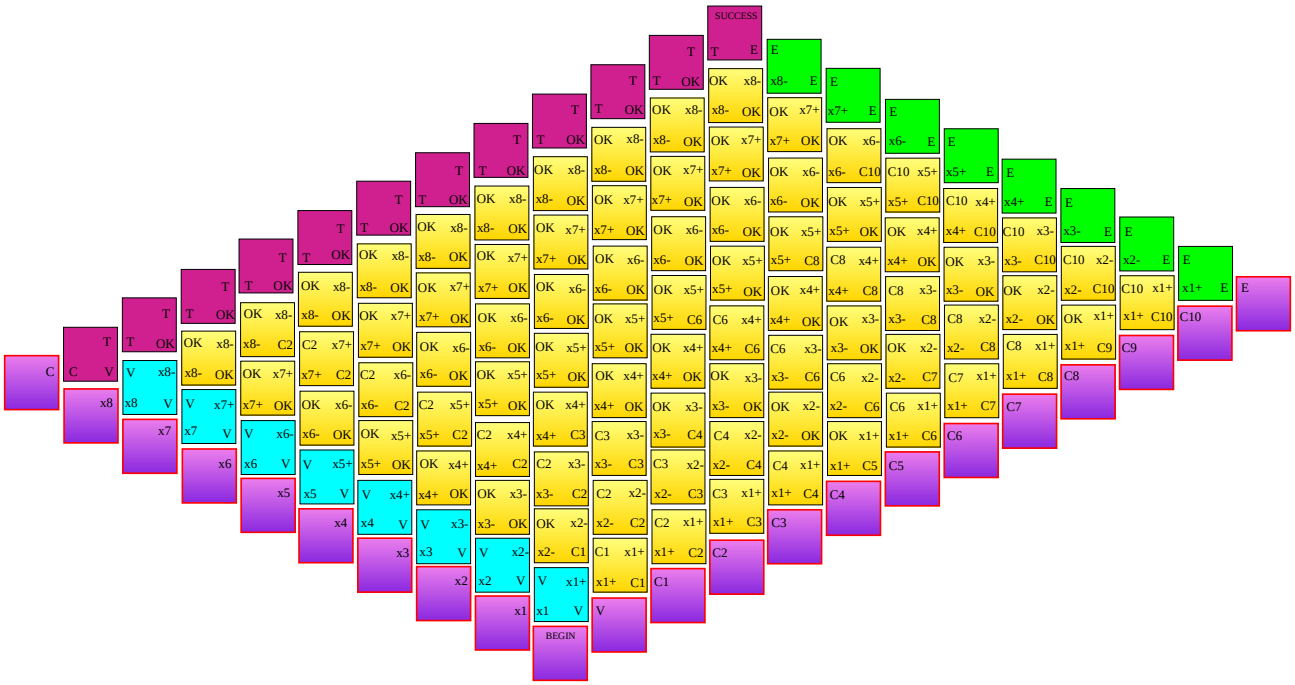


Figure 4: A successful DNA assembly for 3CNF-SAT instance.

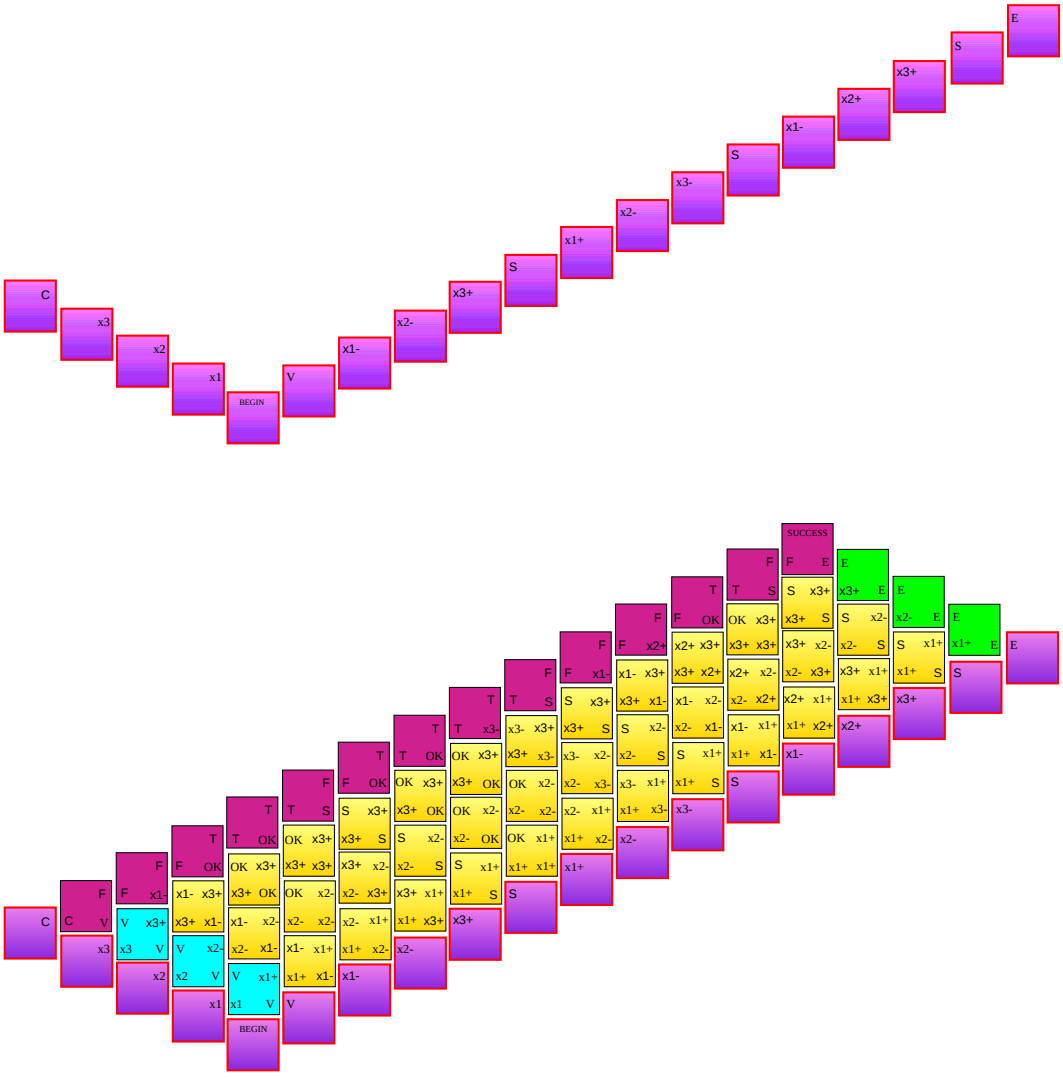


Figure 5: Input tiles and a successful assembly computation for design B.