Constraint Satisfaction Problems on DNA Strings

Luca Bortolussi[1] and Andrea Sgarro[2]

[1] Dept. of Maths and Computer Science, University of Udine, 33100 Udine, Italy.
bortolussi@dimi.uniud.it

[2] Dept. of Maths and Computer Science, University of Trieste, 34100 Trieste, Italy.
sgarro@units.it

Abstract. We explore the possibility of designing a constraint-based algorithm for constructing subsets of DNA words satisfying given constraints (the so-called DNA word design problem). In this direction, we use symbolic representation of sets of strings, and define a propagation algorithm on these representations. Some preliminary results are presented, together with several open problems.

Keywords: *DNA word design, Code Construction, Constraint Programming.*

1 Introduction

In the last ten years, a new computational paradigm emerged from a very uncommon place, i.e. wet labs of biologists. The fact that DNA contains all the basic information necessary to build very complex living organisms convinced Adlemann that it could also be used as a computational entity. In his milestone paper of 1994 [1], he proposed a computational model based on very simple manipulations of DNA that can be performed in a wet lab. This model is Turing-complete and bases its power on the massive parallelism achievable by using DNA. Moreover, one of the basic operations performed is the hybridization of complementary DNA strings. Specifically, DNA strings are oriented strings over the alphabet $\Sigma = \{a, c, g, t\}$, where *a*-*t* and *c*-*g* are complementary letters. Two of such strings are said to be complementary if they have the same length and if one can be generated by reversing the other and complementing each of its letters. Physically, complementary DNA strings can hybridize, i.e. they can attach one to the other, forming the famous double helix. Actually, hybridization can occur also between strings that are not perfect complements, but close to it. In DNA computations, data is coded by short strings of DNA in such a way that hybridizations occurring determine the output of the "algorithm" [8]. Therefore, one of the main concerns is to avoid that "spurious" hybridizations occur, leading straight to the so-called *DNA word design* problem.

DNA word design (cf. [7]) consists of identifying sets of DNA strings of a given length, called *DNA codes*, satisfying some constraints, usually that two of them have Hamming distance and reverse complement Hamming distance greater than a certain threshold. Formally, given $x, y \in \Sigma^n$ and letting y^{RC} be the reverse complement of y, the *reverse complement Hamming distance* between x and y, $d_H^{RC}(x, y)$, is defined as the Hamming distance between x and y^{RC}, $d_H^{RC}(x, y) = d_H(x, y^{RC})$, where the Hamming distance is the usual one, counting the number of positions where the two strings differ.

In particular, the main concern of DNA word design is to identify maximal set of strings satisfying the above mentioned constraints. There is some theoretical work [5] that gives upper and lower bounds to the dimensions of such codes, and also some algorithms constructing such codes [10], that are based on stochastic local search.

The aim of the ongoing work we are presenting here is to give a constraint-based algorithm to build such sets of strings. We will tackle two different versions, both the optimization problem, i.e. find the maximal code, and the constraint satisfaction problem (CSP), i.e. find a set of a given size satisfying the constraints. At this stage of the work, we have made some working hypothesis that simplify the problem. The main one replaces the reverse complement Hamming distance by the simpler *reverse Hamming distance*. Given two strings $x, y \in \Sigma^n$ and indicating by y^R the reverse of y, this new distance is simply defined as $d_H^R(x, y) = d_H(x, y^R)$. A more detailed discussion of the properties of such distance can be found in [9]. Actually, this assumption is painless, as in [5] it is showed that there is a (constructive) one to one correspondence between these codes and the ones making use of the reverse complement distance.

The main problem we have to face, however, is related to the huge dimension of the search space into play. In fact, reasoning for simplicity with the CSP version of the problem, if we are looking for a code of size m, then we have m variables, whose domain is the space of all strings of length n, of size $|\Sigma|^n = 4^n$. Therefore, not only the space of possible solutions has a dimension of the order of 4^{nm} (which for the reasonable values of $n = 10$ and $m = 50$ is around 10^{300}, a gigantic size!), but also there is the problem of finding a way to (over)represent the feasible domains of the variables during the computation, as direct memorization is out of discussion. To tackle this problem we use a symbolic representation of these feasible sets of strings, pretty much in the line of how boolean functions are represented in symbolic model checking. Though these representations, which make use of direct acyclic graphs, can be exponentially large, they seem to perform quite well in practice. In addition, they automatically allow for an effective propagation algorithm that achieves global consistency. In Section 2 we introduce in more detail such mechanisms.

Needless to say, the enormous dimension of the search space pushes for a theoretical analysis of the problem in order to introduce as many constraints as possible. Moreover, an efficient strategy is also needed to choose the branches of the search tree during its exploration. The solution found so far are presented in Section 3. Finally, in Section 4 we present some results and we discuss the current weak points of this approach, outlining some possible solutions.

2 Symbolic Propagation

There are two versions of the DNA code design problem for strings of length n: the optimization one, looking for maximal size codes, and the CSP, looking for a code of fixed dimension m. Both these problems are parametrical w.r.t. the minimum distance D we impose between two strings belonging to it. This distance is

$$d(x, y) = \min\{d_H(x, y), d_H^R(x, y)\}, \tag{1}$$

i.e. the minimum between Hamming and reverse Hamming distance. In particular, in the CSP, we have m variables whose starting domain is the set Σ^n.

In the Introduction we mentioned that one of the main problems in attempting to use constraint-based methods for DNA code construction is the huge size of the domain of each variable. Therefore, there is the problem of storing somehow the feasible values that can be assigned to variables at every point of the search tree. Moreover, we need a way to perform efficiently the propagation of constraints introduced whenever a variable gets instantiated to an element of its domain. These constraints are of the form $d(X, s) \geq D$, where s is the value of the newly instantiated variable.

The techniques we use to tackle both these problems are taken from symbolic model checking [4]. In particular, we use a compact representation of set of strings by means of a particular kind of direct acyclic graphs (DAG) that resembles closely OBDD [3], and that will be called Generalized Decision Diagram (GDD). The very basic idea is that we can identify a set of strings S with its characteristic function $S = \chi_S : \Sigma^n \to \{0, 1\}$. Then we build a rooted DAG representing this function, where nodes are divided into two categories, terminal and non-terminal. There are just two terminal nodes, one labeled with 0 and one labeled with 1. Every non-terminal, or internal, node is labeled by a number from 1 to n. Every internal node has $|\Sigma| = K$ edges, labeled by the K different letters of the alphabet ($K = 4$ for DNA, but the following description is general). Edges always go from nodes with label i to nodes with label $j > i$ or to terminal nodes. Moreover, there is only one node with label one, and it is the root.

The main property of these graphs is that the concatenation of edge labels of a path from the root to the terminal node 1 represent a string belonging to the set S, i.e. evaluating its characteristic function to 1. Concatenating the label $a \in \Sigma$ of an edge exiting from a node with label i, correspond to assign a to the ith letter of the string being constructed.[3] On the other side, the labels of all paths leading to node zero correspond exactly to strings not belonging to S.

A necessary request in order to make sense out of the previous definition is that these GDD graphs must be in canonical form, where there are no redundant nodes (with all edges pointing to the same node) and no duplicated nodes (two

[3] We adopt the convention that if an edge goes from a node i to a node $j > i$, with $j - i > 1$, all position in the string between $i + 1$ and $j - 1$ are set to a wild character $*$, so that a path corresponds more precisely to a string in the augmented alphabet $\Sigma \cup \{*\}$.

nodes with the same label and with edges pointing to the same nodes). This form is unique and can be computed efficiently. Proof and algorithm are a straightforward adaption of the classical ones in [3].
The crucial feature of these representations is that, though containing an exponential number of nodes in the worst case (w.r.t. the number of levels or of different labels of internal nodes), they usually behave well, and have a small and tractable size. Moreover, there exists an efficient algorithm for constructing the GDD G representing the intersection of the sets accepted by two GDD G_1 and G_2, whose complexity is bounded by the product of the dimension of G_1 and G_2 (but can be made more efficient in practice by implementation tricks, cf. [2]). This algorithm is also a straightforward adaption of that presented by Bryant in [3] in the context of boolean functions.
In Figure 1 we show such a GDD for the set of binary strings of length 6 at Hamming distance $d \geq 3$ from the string `000000`. In general, all GDD representing set of strings at distance (either Hamming or reverse Hamming) D or more from a given string s have a shape similar to this GDD.

We use GDD for keeping track of the feasible domain of each non-instantiated variable at each level of the search tree. Every time we choose a branch in the search tree we instantiate a variable, that is to say, we add a string s to the current code. For that string, we construct the GDD for the set of strings at distance $d(X, s) \geq D$. This GDD is constructed by intersecting the two GDD representing the set of strings at Hamming distance at least D from s, and the set of strings at reverse Hamming distance $d_H^R(X, s) \geq D$. Then this resulting GDD is intersected with the GDD representing the feasible domain before the branching. The outcome of this computation is a GDD representing exactly the domain of the non-instantiated variables, given all the constraints (we have constraints of the form $d(X, Y) \geq D$, for every variable $X < Y$). Therefore, the use of symbolic graphical representations for the the variable domains not only allow a compact representation of exponentially big sets, but also, through the intersection algorithm, achieves a propagation which is *globally consistent*.

The problem with GDD is that there is no guarantee that their size stays small. In theory, some of them can have size exponential in n. Luckily, the experimental results run by combining together GDD representing constraints of the form $d(X, s) \geq D$ show that these intersections behave well, and never explode combinatorially. In particular, the GDD corresponding to the constraints introduced by the addition of a single string to the code all have the same, fixed, size and shape.[4] When we start combining these GDD together, to represent the intersection of the corresponding sets, the size of the GDD product rises, while the number of strings belonging to the intersection decreases. After combining a certain number of basic GDD, dependant on the length of the strings, the dimension of the intersection becomes sufficiently small in order to have a compact representation. In every case, the dimension of the bigger GDD in this process remain quite small. We conjecture that this depends on the fact that the

[4] The complementary sets of Hamming spheres, for a fixed threshold, are all isomorphic.

intersection of complementary sets of Hamming spheres, all with same radius, has a sufficiently high internal structure, and thus can be compressed efficiently.

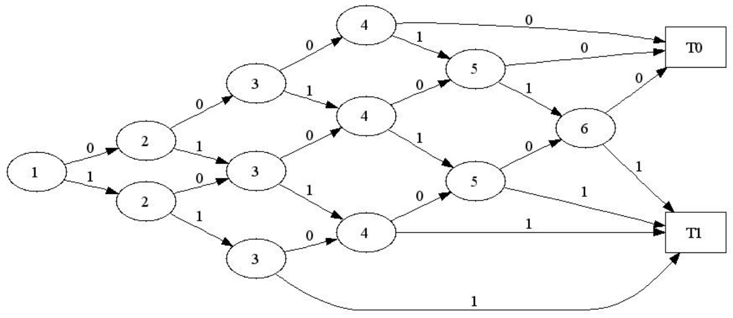

Fig. 1. A generalized OBDD representing the set of binary strings at Hamming distance $d \geq 3$, from the string 000000. Note that or the binary alphabet, GDD are exactly OBDD.

3 More Constraints and Heuristics

The code construction problem has many symmetries and, due to the big size of the search space, we have to break as many as we can. First of all, each set S can appear in all its $|S|!$ permutations. To avoid this, we have to introduce the lexicographical order between strings, and to impose the constraints $X_i < X_j$ for $i < j$.
In addition, given a code $\mathcal{C}$, there are a lot of equivalent codes obtained by transforming $\mathcal{C}$ using isometries of the string space, i.e. functions that are distance-preserving. In the case of our distance (1), these transformations must preserve the relation between conjugate indexes, i.e. couple of indexes whose sum is $n+1$ (for strings of length n). This is due to the fact that these couples of indexes are connected by the reverse transformation.
To break these symmetries, we should introduce constraints guaranteeing that a code $\mathcal{C}$ appears in just one possible way, i.e. forbidding all subsets $\mathcal{C}'$ isometric to $\mathcal{C}$. However, the identification of such set of constraints is still an open problem. What we do for now is fixing the minimum reverse Hamming distance of a word in the code from itself, and then set the value of the first variable to the smallest string in the lexicographical order with such a property. Moreover, if this minimal distance is $2K$ (reverse Hamming distance is always even[5]), then we impose

[5] This is very easy to see: consider, for simplicity, a string of even length, say $2n$, and write it uv, with $|u| = |v|$. Then $d_H^R(uv, uv) = d_H(uv, v^R u^R) = d_H(u, v^R) + d_H(v, u^R) = d_H(u, v_R) + d_H(v^R, u) = 2d_H(u, v^R)$.

the additional constraints $d_H^R(X, X) \geq 2K$, for all variables X. In this direction, another interesting open question is if every maximal code $\mathcal{C}$ contains a palindromic word, i.e. if the minimum self-reverse distance has to be always zero. This seems plausible, as these strings are the less committing in terms of the constraints they impose. In fact, for a palindromic word x, $d_H(x, y) = d_H^R(x, y)$, for each $y \in \Sigma^n$, hence the sets $d_H(x, y) \geq D$ and $d_H^R(x, y) \geq D$ coincide, and their intersection has the biggest possible cardinality.
At the moment, we do not use a symbolic representation of these added constraints, so our propagation algorithm achieves only a local consistency w.r.t. all constraints into play. Despite this, these constraints are exploited in the algorithmic procedure used to select the next string in a branching point. In fact, to choose a new word, we have to look at the paths terminating to node 1 in the current GDD. This can be done efficiently by doing a depth-first-search traversal of the tree that is the unfolding of the portion of the GDD containing these paths. Essentially, these new constraints become heuristics to prune parts of this tree during its exploration.

Another crucial point is the strategy used in the choice of the next word in a branching point. In fact, we want to find quickly a good solution, in order to make pruning effective. The pruning is executed if the size of the biggest code found is greater than the sum of the dimension of the constructed partial code and the dimension of the set of feasible strings (i.e. the dimension of the set accepted by the GDD, which can be counted simply by traversing it).
The strategy we use for the selection of the next string is based on the observation that in a code with minimum distance D, the minimum distance relative to most of its words is also D. Therefore, we select new words first from the subset of feasible strings at minimum distance from the partial code. Another interesting open question is if limiting the branching to this subset still guarantees that an optimal code will be eventually found.

4 Results and Future Work

We implemented our algorithm in **C**, and consequently run several tests. In particular, we tried to look for maximal codes of words of length between 4 and 10 in DNA alphabet, setting a time limit of one hour.
In all these cases, the algorithm finds very quickly a good solution, i.e. a code of pretty high size. Unfortunately, in one hour it never finishes the exploration of the search tree, even for strings of length 4. Moreover, it never finds a better solution than the initial one, even if it discovers other codes of the same size. Some results can be found in Table 1. Unfortunately, there is no literature about code construction w.r.t. distance 1, so we cannot compare easily our method with other approaches.

We also compared our program with $CLP(FD)$ of SICStus Prolog for the easier task of constructing Hamming codes. The results of the SICStus computations are taken from Dovier et alt. [6], and some comparisons can be found in

string length	dist. threshold	size of the solution found	time
4	2	40	0,18 sec
4	3	7	0,03 sec
6	3	59	99,00 sec
6	4	19	0,12 sec
6	5	6	0,03 sec
8	4	113	16,19 sec
8	5	30	4,57 sec
8	6	10	2,47 sec
10	6	58	35,10 sec
10	7	19	52,20 sec
10	8	9	105,18 sec

Table 1. Size of the best code found and time needed to find it, for different string lengths and thresholds for the simplified DNA word design problem.

Table 2. Here we can see that our engine runs around 50 times faster than the one of SICStus Prolog, though it preserves the same pattern of performance: where SICStus fails to find an answer in reasonable time, also our program suffers the same problem.

Taking a closer look to the behaviour of the search, we discovered that the main problem is that the pruning is not very effective. This means that the algorithm has to go very deeply in the tree to realize that a branch cannot contribute with a bigger code than the one found so far. Therefore, to create a more powerful pruning procedure, we need to work out a more clever estimate of the maximum code size, given a partial one.
In addition, the size of the actual search tree is too big to have any hope of exploring it all. The only chance to reduce its size is to find other constraints imposing the uniqueness of the maximal code. The combination of these new constraints and of more powerful pruning heuristics may be able to shrink the

string length	distance threshold	size of the code	existence of a solution	time in SICStus	time in our engine
6	3	8	Y	0,0	0,0
6	3	9	N	562,49	0,5
8	5	4	Y	0,0	0,0
8	5	5	N	3,85	0,03
10	3	64	Y	5,71	0,22
10	5	8	Y	0,05	0,01
10	5	9	Y	668,00	14,40

Table 2. Comparison of the performances between SICStus Prolog and out engine for the construction of Hamming codes.

portion of the tree to be visited to a small enough size. It is not clear, however, if, even with such additions, the algorithm would finish the exploration in a reasonable running time or if simply the task of using enumeration procedures for solving code construction problems is hopeless.

Nevertheless, there is a different and promising direction which we are currently beginning to explore: we are abandoning the target of a complete exploration of the search tree, and trying to integrate some stochastic ingredients allowing to visit just a small portion of the space, characterized by having an high probability of containing the optimal solution. Similarly, we could try to mix the branch and bound schema with local probabilistic search procedures. In this way, we may be able to create an algorithm giving good codes (even if not necessarily the optimal ones) in a reasonable time.

References

1. L. Adlemann, "Molecular Computations of Solutions of Combinatorial Problems". *Science*, Vol. 266, pp. 1021–1024, November 1994.
2. K. Brace, R. Bryant and R. Rudell, "Efficient implementation of a BDD package". *Proc. of the 27th ACM/IEEE conf. on Design Automation*, pp 40–45, 1991.
3. R. Bryant, "GraphBased Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, Vol. C35, No. 8, pp. 79–85, August 1986.
4. E. Clarke, O. Grumberg and D. Peled, *Model Checking*, MIT Press, 2000.
5. A. Condon, R.M. Corn, A. Marathe, "On Combinatorial DNA Word Design". *J. Computational Biology*, Vol. 8:3, pp. 201-220, 2001.
6. A. Dovier, A. Formisano, E. Pontelli, "A comparison of Constraint Logic Programming over Finite Domains and Answer Set Programming in tackling hard combinatorial problems", http://www.di.univaq.it/ formisano/CLPASP/.
7. G. Mauri, C. Ferretti, "Word Design for Molecular Computing: A Survey", *9th Int. Workshop on DNA Based Computers, DNA 2003*, 37-46 (electronic edition), 2003.
8. N. Pisanti, "A survey on DNA computing". *EATCS Bulletin* No. 64, pp. 188–216, 1998.
9. A. Sgarro and L. Bortolussi, "Codeword Distinguishability in Minimum Diversity Decoding". *Submitted to IEEE Trans. on Information Theory.*
10. D. Tulpan H. Hoos and A. Condon, "Stochastic Local Search Algorithms for DNA Word Design". *8th Int. Workshop on DNA Based Computers* , 2003.