University of Southern Denmark

DNA-templated synthesis optimization

Hansen, Bjarke N.; Larsen, Kim S.; Merkle, Daniel; Mihalchuk, Alexei

Published in:
Natural Computing

DOI:
10.1007/s11047-018-9697-7

Publication date:
2018

Document version
Accepted manuscript

Citation for pulished version (APA):
Hansen, B. N., Larsen, K. S., Merkle, D., & Mihalchuk, A. (2018). DNA-templated synthesis optimization. *Natural Computing*, *17*(4), 693-707. https://doi.org/10.1007/s11047-018-9697-7

Terms of use
This work is brought to you by the University of Southern Denmark through the SDU Research Portal.
Unless otherwise specified it has been shared according to the terms for self-archiving.
If no other license is stated, these terms apply:

- You may download this work for personal use only.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying this open access version

If you believe that this document breaches copyright please contact us providing details and we will investigate your claim.
Please direct all enquiries to puresupport@bib.sdu.dk

Download date: 26. Oct. 2020

Natural Computing manuscript No.
(will be inserted by the editor)

DNA-Templated Synthesis Optimization

Bjarke N. Hansen · Kim S. Larsen · Daniel Merkle · Alexei Mihalchuk

Received: date / Accepted: date

Abstract In chemistry, synthesis is the process in which a target compound is produced in a step-wise manner from given base compounds. A recent, promising approach for carrying out these reactions is DNA-templated synthesis, since, as opposed to more traditional methods, this approach leads to a much higher effective molarity and makes much desired (sequential) one-pot synthesis possible. With this method, compounds are tagged with DNA sequences and reactions can be controlled by bringing two compounds together via their tags. This leads to new cost optimization problems of minimizing the number of different tags or strands to be used under various conditions. We identify relevant optimization criteria, provide the first computational approach to automatically inferring DNA-templated programs, and obtain efficient optimal and near-optimal results, and also provide a brute-force integer linear programming approach for complete solutions to smaller instances.

Keywords DNA-Templated Synthesis · Optimization · Trees · Graphs · Cheminformatics

Mathematics Subject Classification (2000) 68 · 92 · 68W40 · 92E10

A conference version of this paper was presented at DNA23 [10]. This paper differs substantially from the conference version: We provide proof of all theorems and have added a section on brute-force computation using integer linear programming. Furthermore, we present an empirical evaluation for the inference of DNA-templated programs with respect to different optimization criteria and we employ a large set of synthesis plans in order to analyze the solution space of the underlying optimization question. The second and third authors were supported in part by the Danish Council for Independent Research, Natural Sciences, grants DFF-1323-00247 and DFF-7014-00041.

B. N. Hansen
Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark

K. S. Larsen
Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
E-mail: kslarsen@imada.sdu.dk
ORCID: 0000-0003-0560-3794

D. Merkle
Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
Tel.: +45 6550 2322
E-mail: daniel@imada.sdu.dk

A. Mihalchuk
Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark

1 Introduction

The first time DNA was used to execute an algorithm in order to solve a combinatorial optimization problem dates back to 1994. In [1], Adleman demonstrated how a small instance of the Hamiltonian Path Problem could be solved using DNA sequences. Since then, DNA nanotechnology has been used as a powerful tool for a wide variety of research and engineering questions. Examples include polyhedral mesh rendering, where DNA sequences are designed such that they fold into predefined complex 3-dimensional structures [3], and design of DNA-based molecular motors that can be used to transport cargo molecules [20].

Appealing features of DNA-based designs is their programmability, the inherent concurrency, the predictability, and the fact that DNA sequences are relatively cheap and easy to synthesize. The number of new approaches utilizing DNA-based chemistry as a source for the discovery and the design of novel drug-like molecules has increased rapidly in recent years [8]. Most major pharmaceutical companies have already started utilizing this technology.

DNA-based chemistry approaches include a method referred to as DNA-templated organic synthesis [14], where the goal is to synthesize an organic compound in a step-wise

manner. In an individual step of a synthesis plan [13], either two compounds are combined (affixation reaction) or a single compound is modified (cyclization reaction). This information can be captured in a rooted unary-binary tree, though often cyclization reactions can be ignored from a combinatorial point of view, making the tree binary. Chemists are aiming at *efficient* synthesis (the yield of all reactions and therefore the yield of the overall process should be high) and (sequential) *one-pot* synthesis (for instance, avoiding complicated separation and purification processes based on contaminating compounds that require subsequent extraction of a specific product from a mixture of compounds).

In DNA-templated synthesis, the base compounds are "tagged" with DNA sequences. These tags are used to bring the compounds in close vicinity (and thereby react). This is done by adding a complementary DNA strand, called an instruction strand, which is a concatenation of the complementary strands of the two tags that are attached to the base compounds. In contrast to classical synthesis approaches, DNA-templated synthesis allows for much lower concentrations of reactants due to the tagging, which leads to a dramatically increased effective molarity. We refer to [14] and [9] for in-depth reviews and specifically [15] and [12] for examples of successful, non-trivial, multi-step DNA-templated molecule syntheses.

The synthesis tree together with a specification of how to tag the base compounds and according to which topological ordering of the tree the reactions should be carried out defines a so-called DNA-templated program. While high-level formalisms for DNA computational structures have been investigated before [17,4], there are no prior attempts to automatically inferring DNA-templated programs based on a given synthesis tree.

In [2], graph rewriting techniques have been used for verifying correctness of given DNA-templated programs, but programs neither were automatically inferred nor optimization questions answered. With careful choice of tagging and topological ordering, it is possible to use the same tags and strands repeatedly, which leads to the optimization problems we consider. To avoid unintended interference, tags and strands that should be different must be some minimum edit distance away from each other. If one uses too many different tags or strands, these must be made longer in order to obtain this, leading to higher production costs.

Another cost stems from the tagging of chemical compounds, which is a somewhat sophisticated chemical procedure. Designing specific DNA sequence strands for tags requires solving optimization problems which consider thermodynamic and kinetic features. Any additional potential tag interaction increases the complexity. While it is interesting to minimize the use of different tags and strands in general, it is also interesting just to minimize the number of different tags used on the base compounds. Specifically in situations where a wet-lab design fails, it is favorable to be able to limit the specific failure reason. This error-finding could be easier in cases where only a few different tags are used.

We present i) optimal or near-optimal methods for minimizing the number of strands, ii) a somewhat more involved method for minimizing the number of strands and subsidiarily the number of tags, iii) a method for minimizing the number of strands when only two different tags are allowed on base compounds, but longer programs using blocking are allowed, iv) a generic ILP formulation of the optimization problems, and, finally, v) an empirical evaluation.

The techniques considered before the ILP formulation are specialized, targeted at particular optimization criteria, and they are very efficient in terms of runtime complexity (close to linear time), enabling us to solve the problems being addressed for any size problem which could meaningfully be considered in practice.

In contrast, the ILP formulation is much more flexible and one can in principle make variations, optimizing for a (weighted) combination of different goals, etc. On the negative side, ILP is an NP-hard problem, so the runtime complexity of whichever ILP solver is used is exponential time on worst case instances for ILP. Our instances may not be the hardest, but in general, it is not easy to determine the runtime complexity of concrete ILP instances. Since flexibility is sometimes desirable, we include an example of how this method can be employed, such that this is an option for problems small enough that the execution time is not prohibitive.

2 Modeling DNA-Templated Synthesis

The goal of this section is to present a model for DNA-templated synthesis such that we can work with these issues in a combinatorial manner. We identify some basic operations and restrictions on how these can be applied, with the goals in mind. We would like to emphasize that we do not make any simplifying assumptions, preventing our solutions from leading to programs that can be realized chemically. However, there could be other choices of computational units and goals, and our focus is on presenting an initial model that is as simple as possible while still capturing the fundamental chemical intricacies. Our description will lead to a definition of the input, available operations, constraints, and a number of optimization objectives.

From a chemist, we get *a synthesis tree*, which we assume is binary, where the *leaves represent compounds*. We refer to these as the *base* compounds. The tree can be interpreted as a recipe in the following manner. Each leaf of the tree represents an existing base compound. Now, we bring compounds to react in an order respecting the tree structure.

Thus, first the compounds corresponding to two leaves are made to react, resulting in a new compound, which we refer to as an *intermediate* compound, and we represent it by the parent node of the two interacting compounds represented by its children. We keep going until we reach the root, and have at that point produced our final *target* compound. The order of combining the compounds should simply be a topological ordering of the tree. We draw the trees with the root at the bottom, as it is usually done for synthesis trees.

We detail the operations below. Our textual description is complete and self-contained, but it might be helpful to refer to the appendix for an example program, and the operations are summarized in Table 1. In order for two compounds to react, they must be in close proximity, and two compounds do not react if they are distant enough. To obtain proximity, the compounds are equipped (tagged) with DNA sequences, and the compound is at one of the two ends of the sequence, i.e., either at the 5'-end or the 3'-end of the sequence. We refer to such a DNA sequence on a compound as a *tag* and choose an orientation so that we can refer to the left and right ends of a tag. In our illustrations, tagged compounds will always be depicted such that the 5'-end corresponds to the right end of a tagged sequence, and the 3'-end corresponds to the left end of a tagged sequence. Assume x and y are the tags of compounds X and Y, respectively, and X is at the right end of x and Y at the left end of y. If we add the *complementary* strand of the concatenation of x and y, denoted $\overline{xy}$, x and y will attach to the $\overline{x}$-part and $\overline{y}$-part of $\overline{xy}$, respectively, bringing X and Y close together and the reaction of X and Y takes place. We refer to such a strand as an *instruction* strand and the process as a *react* operation. Instruction strands are always depicted such that the 5'-end of the strand corresponds to the left end of a sequence, and the 3'-end corresponds to the right end of a sequence. Consistent with the common arrow notation for DNA, the 3'-end of tags and strands will be depicted with an half-arrow in all figures. In the above, and in the rest of the paper, when we refer to a strand, it can always be thought of as the concatenation of two tags. The intermediate compound resulting from a react operation will be tagged with either x or y in a deterministic fashion decided by the compounds, i.e., along with the synthesis tree, a chemist will tell us, for every internal node, which of the two tags from the child nodes will be the tag of the produced intermediate compound. We say that the node *inherits* the tag from the child in question and we may use a bold edge to indicate this. This annotated tree forms our input from the chemist. Note that the compounds and what they become when they react is not important to us; only the tags (and how they are attached) and strands are relevant to our computation.

Concatenated DNA strands are denoted as the sequence of the corresponding strand variables, where the string representation of the concatenation of variables is always given from left to right (which is not necessarily from the 5'-end to the 3'-end), i.e., $\overline{xy}$ denotes the complementary strand for the strand xy.

After a reaction has been carried out using the $\overline{xy}$ instruction strand, a complementary *release* strand, xy, is added to release the compound, and, thus, prepare for further reactions.

We disallow simultaneous releases, since they lead to a low overall yield as we explain now. Releasing two compounds using $\overline{xy}$ implies that one released compound must be tagged with an x and the other with a y. Otherwise (that is, if both compounds have the same tag), we cannot control subsequent operations. But this implies the presence of free-flowing y strands and x strands from the first and second reactions, respectively. These may attach to any later $\overline{xy}$ strand, resulting in a reduced overall yield.

A final chemical possibility we shall use as an operation in one section is the ability to temporarily *block* a compound. A compound tagged with a strand x can be blocked by adding a strand $\overline{xy}$ or $\overline{yx}$, and can be released again in the same manner as described above.

We use blocking in Section 3, but otherwise simply delay the release of compounds while working on others, with the aim of producing a program for sequential *one-pot* synthesis. Compounds, corresponding to the leaves of our input tree, may be added gradually, but we do not allow ourselves to produce compounds corresponding to subtrees separately and add them later.

From a practical perspective it should be mentioned that a blocking operation requires additional blocking strands to be added during the one-pot synthesis. In a real-world setting, the correct molarity of strands to be added needs to be chosen such that the amount of added blocking strands of a certain type will indeed block all tagged compounds that should be blocked, but also such that no blocking strand is unused, as this might lead to subsequent unwanted inferences. In order to circumvent this precision requirement, it might be desirable to completely avoid the use of blocking. Furthermore, even though blocking theoretically leads to inert waste, it might be favorable to reduce the amount of waste created by blocking, as it might interfere in an unexpected manner in a practical setting.

Our computational choices are the following. Given the annotated synthesis tree, we must decide on tags for the leaves and a topological ordering, including when to add, when to release, and in one algorithm also when to block and which strand to block with. Recall that given tags on the leaves, the annotation determines the tags on internal nodes. Since we most often use delayed release to avoid interference, we will frequently label internal nodes with the instruction strand, i.e., the sequence of two tags. The tag attached to the intermediate compound produced at that node is always one of the tags the strand consists of, and which

one it is, is determined by the inheritance information provided by the chemist.

In summary, a program is a sequence of operations (tag, react, release, block), where the operation *tag* attaches a specified tag to a base compound, *react* combines two intermediate compounds, *release* releases the resulting intermediate compound, and *block* blocks a compound. To be chemically feasible, left and right input compounds to any react operation must have the compound placed to the right and left, respectively, the react operations must form a topological ordering of the tree, compounds (unreleased as well as possibly blocked) must be released (unblocked) before they are used again, any block operation must use a strand matching the compound tag to the left or right, all unreleased (and blocked) compounds in the pot at any given time must be unreleased (and blocked) with unique (at the time) strands, and if there are compounds in the pot with the same tag, all but one must be unreleased or blocked.

This is implied by the above, but just for emphasis, we cannot use strands of the form *xx* in a controlled process, so if we use τ different tags, we have at most $\tau(\tau-1)$ different strands at our disposal.

We illustrate some of these restrictions now, using the smallest possible interesting synthesis trees. First note that because compounds are at one end of a tag, we cannot have an unreleased compound with an $\overline{ab}$ strand while using *ba* at the root of the other subtree. This is because when we release using *ab*, then (without loss of generality) the released compound is tagged with *a* and the compound is at the right end. Thus, later, it must react with a compound tagged with a *b* where the compound is at the left end. Thus, the strand from that subtree would have to have the form *xb* for some *x*; see Fig. 1.

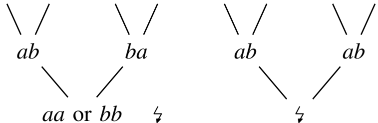


Fig. 1 Illustration of disallowed strand assignments. Left: Using strands *ab* and *ba* for two children requires the parent to be assigned either the strands *aa* or *bb*, which will result in a reduced overall yield, as with a probability of 50%, the corresponding compounds do not get in close proximity and therefore will not react. Right: Assume one subtree is already computed and the compound has to be unreleased with the complementary strand $\overline{ab}$. The corresponding unrelease needs to make the waste inert with $\overline{a}$ or $\overline{b}$, depending on which tag is now flowing freely in the pot. However, due to the disallowed simultaneous release of the other subtree, the release operation of the last of the two subtrees would accidentally make tagged compounds inert.

The reader may have wondered if the reverse sequence of *x* is any different from *x* in a pot, or if *xy* could interfere with *yx*. Starting with the latter, breaking the sequences into their nucleotides, $\alpha_1\alpha_2\cdots\alpha_{n_\alpha}\beta_1\beta_2\cdots\beta_{n_\beta}$ is different from $\beta_1\beta_2\cdots\beta_{n_\beta}\alpha_1\alpha_2\cdots\alpha_{n_\alpha}$, and they are not the reverse of each other. Obviously, *x* cannot be distinguished from its reverse sequence in a pot. However, compounds are attached to one of the ends, so everything has an orientation.

Finally, to give a clean initial presentation, we do not consider the option of adding multiple strands simultaneously. Allowing this would not add expressive power and for most problems where the objective is to use for smallest number of different strands, it is counter-productive. However, in a lab, it could be desirable to know when this is an option, since this could speed up the process. One could also lift the restriction of sequential one-pot synthesis. However, since this would lead to a multi-criteria problem, we prefer to focus on the cleaner sequential one-pot problem.

Some of the algorithms in this paper and graphical illustrations of the chemical processes can be inspected via a prototype implementation [11].

3 Minimizing the Number of Tags

In this section, our objective is to minimize the number of tags used on base compounds (the leaves), and as our second priority, we want to minimize the total number of tags used.

It turns out that, with appropriate blocking, it is always possible to arrive at a program using only two tags on base compounds, and clearly, for any two sibling leaves with the same parent, the tags must be different. We refer to the two tags as *a* and *b*. Using the following recursively defined function, $\left\lceil \frac{\textsc{Mnt}(\textsc{Root},0,0)}{2} \right\rceil$ will compute the minimum number of additional tags needed to block intermediate compounds when the basic compounds are tagged using only *a* and *b*.

In the formula below, computing the number of additional tags for a tree t, we let t_a and t_b denote the subtrees of t, where t_a, respectively t_b, denotes the subtree representing the compound tagged with an *a*, respectively *b*.

We keep track of tags used in strands together with *a* and with *b* separately, using c_a and c_b as counters. Now, we define $\textsc{Mnt}(t,c_a,c_b)$ to be

$$\begin{cases} \max(c_a,c_b) & \text{if } t \text{ is a leaf} \\ \min\left(\begin{array}{l} \max\left(\begin{array}{l}\textsc{Mnt}(t_a,c_a,c_b),\\ \textsc{Mnt}(t_b,c_a+1,c_b)\end{array}\right), \\ \\ \max\left(\begin{array}{l}\textsc{Mnt}(t_a,c_a,c_b+1),\\ \textsc{Mnt}(t_b,c_a,c_b)\end{array}\right) \end{array} \right) & \text{otherwise} \end{cases}$$

We discuss correctness and the derived program in the following. The DNA program example in Appendix B is designed to demonstrate many features at once, and also shows how blocking can be employed. First, we decide arbitrarily

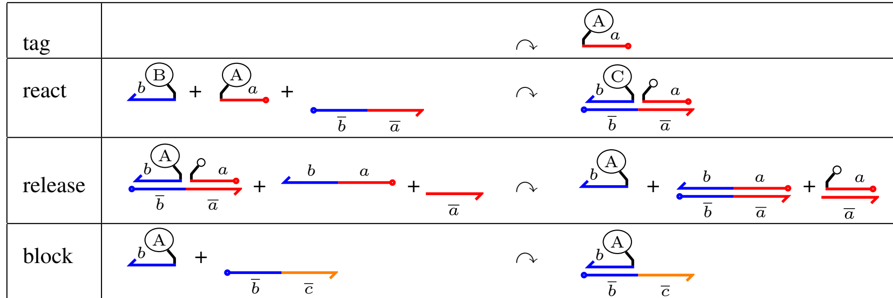


Table 1 Operations of a DNA-templated program: note, that i) the tag operation allows for attaching the compound to the left or right end of the tag, ii) the inheritance for the react operation is given as input from the chemist, iii) the release operation assumes an addition of complementary tags in order to handle waste, iv) the blocking operation can bind the tagged compound to the left or right part of the added strand.

between a and b for the final tag on the target compound that the root represents. If we use only the two tags a and b on base compounds, then we can determine all tagging recursively, since the chemist has informed us, for each node in the subtree, from which child we inherit the tag, i.e., if a node has a given tag, then a specific child of that node must have the same tag, and then the other child must be given the other tag (of the two tags a and b).

Since compounds have one of only two tags, any reaction involves both tags, so anything else in the pot must be blocked. In algorithms to be presented later, leaving them unreleased can also be an option, but in this particular case with only two tags on compounds, this would lead to the disallowed simultaneous release; see the earlier Fig. 1.

As a consequence, for any node with two non-leaf subtrees, we must decide which subtree to synthesize first, and then block while we work on the other subtree. In the subtree we synthesize first, we must block other compounds (corresponding to subtrees) recursively. We find the best subtree to block using the minimization in the formula above. The first entry in the minimization corresponds to first synthesizing and then blocking the subtree t_a. This requires no further resources while synthesizing that subtree, but while later synthesizing t_b, the compound from t_a must be blocked using a tag that has not been used for blocking subtrees on the way from the root to this node. Actually, when using some tag x to block a, for instance, this can be done (unconstrained) as ax or xa. Thus, each such tag x can be used twice, which accounts for the fraction $\frac{1}{2}$ in the final result, $\left\lceil \frac{\text{MNT}(\text{ROOT},0,0)}{2} \right\rceil$.

The best values can be computed using dynamic programming. If the tree is of height h, then each of the variables a and b in the expression can take on at most h different values, so if the tree has size n, then $O(nh^2)$ is an upper bound on the number of values to be computed and each value in a given node can be computed in constant time from values in the node's subtrees, so $O(nh^2)$ is also an upper bound on the computation time. A program can easily be extracted from the computed values by simply checking if the various minima are obtained from the left or right. An example program is shown in the appendix.

4 Minimizing the Number of Strands

In this section, we consider the problem when it is undesirable to use blocking, so that is disallowed, and our objective is to minimize the number of strands used. We allow for an arbitrary number of tags. As any instruction strand requires a unique complementary release strand, they will not be counted separately. It turns out that it is necessary and sufficient to use $\mathscr{S}(t)-1$ different strands, where $\mathscr{S}(t)$ is the (Horton-)Strahler number [19] of the synthesis tree t. Referring to the previous section, where we restricted ourselves to only using two different tags on base compounds, the Strahler number many strands would not in general be sufficient. The result in this section is accomplished without using blocking.

Definition 1 The Strahler number $\mathscr{S}(t)$ of a binary tree t is defined as follows: If t is a leaf, then $\mathscr{S}(t)=1$, and if t has two subtrees t_l and t_r, then

$$\mathscr{S}(t)=\max(\min(\mathscr{S}(t_l),\mathscr{S}(t_r))+1,\max(\mathscr{S}(t_l),\mathscr{S}(t_r)))$$

$\mathscr{S}(t)$ is also referred to as the register number, i.e., the minimum number of registers required for evaluating a given arithmetic expression. The first algorithmic approach to this seems to be from [6], with extensions and variations continuing in many different directions, inspired by compiler optimization problems, considered in [16, 18, 7], among other texts.

Proposition 1 *The number of different strands needed to treat a binary synthesis tree t, in the worst case, to facilitate a sequential one-pot synthesis equals the number of registers needed to evaluate an arithmetic expression with black-box operators.*

Proof At all times, in the worst case, all compounds in the pot other than the interacting pair must be blocked in order to prevent unwanted reactions.

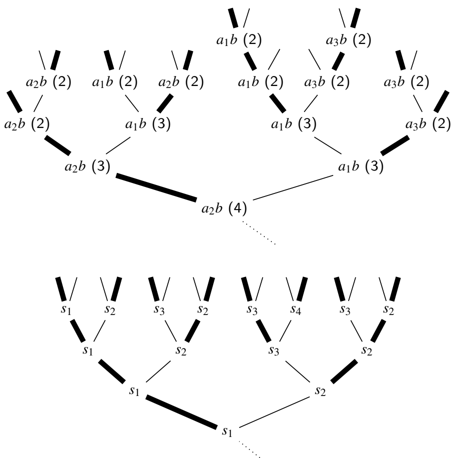


Fig. 2 Top: Illustration of the labeling algorithm that uses $\mathscr{S}(t)-1$ many strands $a_1b, a_2b, \ldots$. Note that this is only one of the possible labelings, since strands are simply chosen from an available set, though we have consistently chosen the smallest indexed a_i available. The Strahler number is given in parenthesis. Bottom: Illustration of the labeling algorithm for complete binary trees: $s_1, s_2, \ldots$ is an antipath of strands, inheritance of tags is illustrated by bold lines.

Each blocked compound needs a different strand. When the compound is unblocked, that strand can be reused.

For arithmetic expressions, we use the term black-box operators to indicate that no algebraic properties can be utilized, such as associativity or other properties which might enable one to rewrite the arithmetic expression to one with a different parser tree than the original.

Thus, the situation is symmetric. Each temporary value that has been computed must be stored in a register. When the value is used as an argument to the parent operation, the register can be reused. □

We are given a synthesis tree and information regarding from which child a node inherits its tag. To explain the tagging, it is easiest for us first to reorder the subtrees so that tags are inherited from subtrees according to a specific pattern. By a *layer* in a tree, we denote all the nodes of the same distance from the root. Given the synthesis tree, we order the subtrees such that when considering any layers from the left to the right, the tag is inherited alternately either from the left or from the right child, and we start by inheriting from the left; see bold edges in Fig. 2.

Now, we explain how we label each node in our synthesis tree, excluding the leaves that contain the base compounds. For the labeling, we use the set

$$I = \{a_1b, a_2b, a_3b, \ldots\}.$$

This set contains strands that have pairwise different tags as their first parts (a_i) and identical tags as their second parts (b).

Recursively for a subtree t of the synthesis tree with ordered children as described above, we first compute the subtree with the larger Strahler number. In case of identical Strahler numbers, we choose the left subtree first. The strand assignment is done as follows: In case the subtrees have identical Strahler numbers, the subtree computed first will require a strand for the release operation. This strand cannot be used for any operation in the other subtree. If the Strahler numbers are different, this constraint will not apply. However, in all cases, neighboring operations need to use different strands. During the recursion, we keep track of the set of forbidden strands (this set grows by one element for the right subtree in the case of identical Strahler numbers) and the sibling reaction strand. Note that the constraint for the sibling reaction *only* applies to the sibling reaction. The pseudo-code is given in Alg. 1 and an illustration with an example of the labeling for a tree with Strahler number 4 is given in Fig. 2 (Top). With regards to the number of strands,

Algorithm 1 Strahler Number Strands

Given: Synthesis tree t ▷ ordered children according to text description
  Set $A = \{a_1b, a_2b, \ldots, a_{\mathscr{S}(t)-1}b\}$ ▷ A: set of strands with $|A| = \mathscr{S}(t)-1$

1: **function** AssignStrand(Tree t, Set F, Strand *sibling*) ▷ F: forbidden strands
2:   $t_l, t_r \leftarrow$ LeftSubtree(t), RightSubtree(t)
3:   **if** both t_l and t_r are base compounds (leaves) **then**
4:     choose strand s from $A \setminus (F \cup \{sibling\})$
5:   **else if** one of t_l and t_r is a base compound (a leaf) **then**
6:     $t_x \leftarrow \arg\max_{t_i \in \{t_l, t_r\}} \mathscr{S}(t_i)$ ▷ t_x is the non-leaf tree
7:     $s \leftarrow$ AssignStrand(t_x, F, _)
8:   **else**
9:     **if** $\mathscr{S}(t_l) > \mathscr{S}(t_r)$ **then**
10:       $s \leftarrow$ AssignStrand(t_l, F, _)
11:       AssignStrand(t_r, F, s)
12:     **else if** $\mathscr{S}(t_l) < \mathscr{S}(t_r)$ **then**
13:       $s_r \leftarrow$ AssignStrand(t_r, F, _)
14:       $s \leftarrow$ AssignStrand(t_l, F, s_r)
15:     **else** ▷ $\mathscr{S}(t_l) = \mathscr{S}(t_r)$
16:       $s \leftarrow$ AssignStrand(t_l, F, _)
17:       AssignStrand(t_r, $F \cup \{s\}$, s)
18:   assign s to t
19:   **return** s
20: AssignStrand(t, $\varnothing$, _)

it is clear that the forbidden set F grows with the Strahler number, so if it was not for the temporary restriction given by the sibling, we use $\mathscr{S}(t)-1$ strands. Recall that a leaf (with a compound) has Strahler number one, so the smallest subtree we assign a strand to has Strahler number two. With regards to the restriction, when the number of available strands is at least two, the temporary restriction does not matter, since we still have a strand we can choose. Thus, the only possible problem is when we recur from a tree with Strahler number three to smaller subtrees. If the subtrees have different Strahler numbers, there is no problem, since the restriction is imposed on the smaller one. If they have

the same Strahler number, the sibling restriction coincides with the growing forbidden set, so only one strand option disappears, and the one required strand can be found.

With regards to efficiency, we assume the Strahler numbers are computed first and stored in the nodes. In other words, we are not thinking of $\mathscr{S}(t)$ as a recursive function, which would lead to repeated recomputation of Strahler numbers for subtrees.

With regards to chemical feasibility, siblings have different strands by construction, and b has its compound at the left and the compound coming from the right subtree will always be tagged with b. The opposite holds for the a_is, so the strands listed in the internal nodes indicate instruction strands fulfilling all requirements.

The upper bound just given is the interesting one. The lower bound that $\mathscr{S}(t) - 1$ different strands are necessary follows directly from the equivalent result for arithmetic expressions [7]; it is simply a matter of having to store at least that many intermediate results.

Strahler examples, as the ones produced in this section, can be found in [11].

5 Complete Binary Trees

The two problems of minimizing the number of strands used (Section 4) and minimizing the number of tags used under the constraint that all base compounds are tagged by one out of two tags (Section 3) can both be solved optimally in an efficient manner. In this section, we restrict the topology of the synthesis plan to complete binary trees and present an approach that minimizes the overall number of strands *as well as* bounds the overall number of tags to the optimal, possibly plus one. We accomplish this without using blocking.

The approach will employ so-called antipaths [5], which is a sequence of adjacent edges in a digraph, where every visited edge has opposite direction of the previously visited edge; and we will need some further restrictions defined below.

Definition 2 Given a digraph, let $u_1, u_2, u_3, \ldots$ be vertices representing a path in the equivalent undirected graph (ignoring orientation of edges). The vertices are not necessarily distinct, but for any i, $u_i \neq u_{i+1}$. An antipath in a digraph is a finite sequence of directed edges (u_i, u_j) having one of the following forms:

$$(u_1,u_2),(u_3,u_2),(u_3,u_4),\ldots \text{ or } (u_2,u_1),(u_2,u_3),(u_4,u_3),\ldots$$

That is, every second time repeating the origin of the edge and every second time the destination of the edge.

An antipath is called *return-free* if for any two successive edges (u,v) and (u',v'), $\{u,v\} \neq \{u',v'\}$ and *non-overlapping* if no edge is used twice.

One can view the above as the well-known concept of Eulerian paths in undirected graphs, generalized to antipaths in a directed setting.

In our construction, we need return-free, non-overlapping antipaths as long as possible (each edge will correspond to a strand) from digraphs with as few vertices as possible (each vertex will correspond to a tag).

Theorem 1 *In a complete digraph $G_n = (V,E)$ over $n \geq 2$ vertices, the length of a longest return-free, non-overlapping antipath is $n(n-1)$ if n is odd and $n(n-2)+1$ if n is even.*

Proof Since $n(n-1)$ is the number of edges in a complete digraph, no non-overlapping antipath can be longer. For the even case, we can show a smaller upper bound: An antipath may have a start and an end vertex, but, due to the definition of an antipath, every vertex internal to the path (that is, not the start or end vertex) must appear as the origin vertex an even number of times and as the destination vertex an even number of times. Since, for n even, a vertex has an odd number $n-1$ of neighbors, each internal vertex can appear as the origin at most $n-2$ times and as the destination at most $n-2$ times. Thus, the total number of times vertices appear as origin or destination is at most $n \cdot 2(n-2)+2$, where the additive term of two accounts for the fact that if the start and end vertex of the antipath are different, then each of them can appear one time extra as origin or destination, respectively. Since an edge consists of two vertices, the number of edges is at most half of the number above, i.e., $n(n-2)+1$.

Having established the upper bounds, we show that it is indeed possible to construct antipaths of those lengths.

The proofs are by induction in n and we start with odd n. For $n = 3$,

$$V = \{u,v,w\},\ (u,v),(w,v),(w,u),(v,u),(v,w),(u,w)$$

is a return-free, non-overlapping antipath of length six. This matches the upper bound.

For the induction step, assume the result holds up to some n, and assume we add the vertices u' and v' to G_n to obtain G_{n+2}. Choose a maximum cardinality matching in G_n. It will contain $n-1$ vertices, that is $(n-1)/2$ pairs, since the graph is complete. Since n is odd, exactly one vertex, x, is not part of the matching. For each pair, (u,v), in the matching, we replace the edge (u,v) in the antipath by the following:

$$(u,u'),(v,u'),(v,v'),(u,v'),(u,v),(u',v),(u',u),(v',u),(v',v)$$

This adds eight edges per pair, or four edges per vertex. This results in a return-free, non-overlapping antipath in G_{n+2}. Since, by induction, all edges are included in the antipath for G_n, by these transformations, all new edges, except edges between the two new vertices u' and v', are added to the antipath. To include also the two new edges, we take

the unique vertex x not included in the matching and connect x, u', and v' as in the base case, adding $6 = 4+2$ edges, which can be inserted anywhere in the antipath where x is visited (choosing the right orientation). By this construction and the induction hypothesis, the constructed path has length $n(n-1)+4n+2 = (n+2)(n+1)$.

For the case of n even, assume first that $n = 2$ and $V = \{u, v\}$. Then (u, v) is a return-free, non-overlapping antipath of length one. For the induction step, assume the result holds up to some n. Again, we choose a maximal matching of $n/2$ pairs of vertices, but now in such a way that all pairs belonging to the matching are connected by an edge which is included in the antipath. We extend the induction hypothesis to show that such a matching exists (this is no longer trivial, since not all edges are present in the antipath as it was the case for the odd case). For each pair, (u, v), in the matching, we perform the same transformation as above. For the new complete matching, we choose one pair (u, v) in the matching for G_n, and define the matching for G_{n+2} to be the same, except that (u, v) is removed and (u, u') and (v, v') are added. By this construction and the induction hypothesis, the constructed path has length $n(n-2)+1+4n = (n+2)n+1$. □

As in all the other sections, we are given a synthesis tree and information regarding from which child a node inherits its tag. We reorder subtrees with regards to inheritance as in the previous section.

Separate from the tree structure, assume that we let each tag that we use represent a vertex in a digraph. Thus, a directed edge in the digraph is an ordered pair of tags, which we can interpret as a strand. We choose a longest antipath $s_1, s_2, s_3, \ldots$ in such a digraph, writing them as s_i for the ith strand. The number of tags (equal to the number of vertices in the digraph) we use depends on how long an antipath we need for the construction below. We emphasize that the purpose of the antipath from digraph is only to generate strands for the synthesis tree such that the tags used in the strands have certain properties. Thus, the antipath in the digraph is just strand production machinery for the synthesis tree, which has otherwise no relation to the digraph and antipath.

First, we explain how we label each node in our synthesis tree, excluding the leaves that contain the base compounds. The root is labeled s_1 and, for ease of the definition below, artificially assume that the root has a parent, and that we moved left to get to the root. Moving from the root towards a leaf, we label each node with the same label as its parent (below it in our illustrations) if we move in the same direction as from the grandparent to our parent, and we label it with the next label (index one larger) if we change direction; see Fig. 2 (Bottom). Afterwards, the base compounds in the leaves can be tagged in the obvious manner, tagging the left (right) leaf with the left (right) part of its parent's strand.

From the labeled synthesis tree, we can define the program recursively. For a given node, we first compute the subtree, the root of which has the strand with the smallest index, leaving it unreleased while the other subtree is computed, after which the first subtree is released and the instruction of the node is carried out.

We now argue that the labeling algorithm produces a chemically feasible program. With regards to the reactions, due to the definition of the inheritance, a simple inductive argument establishes that at any node, the two input compounds stem from the left-most and right-most leaves of the subtree of which the node is the root. Thus, the compounds are tagged with the correct orientation for a reaction. With regards to the interference, the definition of the program explicitly states that the subtree, the root of which is labeled with the smallest indexed strand is computed first, and by the labeling algorithm, that strand is not used in the other subtree. Thus, no release operation can unintendedly release more than one compound.

Properties of the labeling of the complete binary tree depend on the properties of the strands, which in turn are produced via antipaths in a complete directed graph. We emphasize that the tree and the complete digraph are separate entities, and when we talk about antipaths, this is only with respect to the digraph used for the strand construction.

Theorem 2 *The labeling algorithm uses the minimal number of strands and at most one more than the minimal number of tags.*

Proof A complete binary tree of height h has Strahler number $h+1$, so we know from Proposition 1 and the discussed results on arithmetic expression evaluation that h is the optimal number of strands. The maximal number of direction changes from the root to the level next to the leaves is $h-1$, so, since the root is labeled s_1, the maximal label index is $1+(h-1) = h$.

Assume that it is somehow possible to make a program using the optimal number of tags τ. Observe that we can make at most $\tau(\tau-1)$ different strands from τ tags, so if τ is the optimal number tags, this must mean that this hypothetical program uses at most $\tau(\tau-1)$ strands.

If we allow for $\tau+1$ tags in our program, we know from Theorem 1 that an antipath of length at least $(\tau+1)(\tau-1)+1$ exists in the complete digraph used for the construction of strands. Since we use the optimal number of strands and $(\tau+1)(\tau-1)+1 \geq \tau(\tau-1)$ for any positive integer τ, the theorem follows. □

We remark that the construction is actually optimal also with regards to the number of tags in many cases. In fact, for all heights up to 25, we know that we are optimal, except for the heights 10–12. An example argument that the method is optimal for height 13 (in fact, the same argument works up

to height 20) goes as follows. We know we need 13 different strands. With 4 tags, we can make only $4 \cdot 3 = 12$ different strands, so 5 tags are necessary for any program, and with 5 tags we can find antipaths of lengths up to $5 \cdot 4 = 20$. Similarly, for height 9 (in fact, down to height 7), we need 4 tags to have enough strands, and with 4 tags, we can make antipaths of lengths up to $4 \cdot 2 + 1 = 9$. It is the slightly limited lengths of antipaths for an even number of tags that prevents us from extending this optimality argument throughout the range 10–12.

Finally, the algorithm runs in linear time. The recursive definition of the longest antipath one can extract from the theorem is constructive and easily implemented in linear time in the number of strands needed for the synthesis tree algorithm, the labeling is a linear-time pre-order traversal, and the extraction of the program is a linear-time depth-first traversal.

6 Brute Force Approach

In this section, we present a generic approach to minimizing the total number of tags used. It is generic in the sense that we base it on some interference information, which could come from any topological order of the synthesis tree, and blocking could be allowed, disallowed, or used in some locations. This represents an example of what we could optimize for; we could just as well use it to optimize for the number of strands used.

This is the main point of this approach, where we use develop an Integer Linear Program (ILP). The methods described earlier are targeted at specific problems. They are extremely efficient, but if one wants to implement a small variation, the development of a correct and efficient algorithm may have to start from scratch. In contrast, ILP-solvers may not solve specific problems quite as efficiently as a targeted solution. However, they are fairly efficient, and have flexibility as their major advantage. One can easily add further constraints, such as, for instance, that a specific unbound tag should never be in the pot at the same time as another specific tag.

We explain how to create an ILP from the interference information. Given a synthesis tree, choosing a topological ordering determines the order in which compounds must be brought to react. Given such a sequence of react operations, we can additionally choose to add blocking. The brute force approach exhaustively enumerates all topological orders of a synthesis tree, and all optional insertions of blockings at all locations. This is of course potentially much more time-consuming than the specific approaches from earlier sections, but on the other hand doable for synthesis trees that are relatively small.

All such programs, obtained from a specific topological ordering and optionally inserted blockings, give rise to some requirements of the form "xy must be different from $x'y'$", where x,y,x',y' are tags and the concatenation of two tags are unreleased instruction strands or blockings. And, as discussed earlier, this requirement boils down to the requirement that x must be different from x' or y must be different from y'. In addition, for all strands xy used in the program, we must have that x is different from y.

Of the n nodes in the synthesis tree, some are leaves, containing a tagged compound. Some compounds may be equipped with the same tag, but, in total, there are no more tags on compounds than there are leaves. Similarly, we may use a tag for blocking at any internal node. Again, some of these may be identical and could also be identical to leaf tags. However, n is an upper bounds on the number of tags used. We can write down our interference information in terms of these tags, letting $(i,j,k,l) \in I$ denote that the concatenation of the ith and jth tags must be different from the concatenation of the kth and lth tags. Similarly, for all strands in our program formed as the concatenation of tags i and j, we let $(i,j) \in S$.

We can now formulate the ILP. As accounted for in the above, there are n potentially different tags, but it may be possible to make some of them identical, depending on the interference information. One way to think of this is that we have n place holders and we have to place a tag in each. One tag may be placed in several different place holders, provided that we do not violate an interference constraints. We use binary variables v_{it} and $v_{it} = 1$ if and only if the ith place holder gets the tag t. The place holder, i, range over $V = \{1,\ldots,n\}$. We also use the set $T = \{1,\ldots,n\}$. Even though $T = V$, we use T to emphasize when we range over concrete tags given to place holders. The ILP is given in Fig. 3.

$$
\begin{array}{llll}
\text{minimize} & v_{max} & & (1)\\
\text{s.t.} & \sum_{t\in T} v_{it} = 1 & \forall i \in V & (2)\\
 & \sum_{t\in T} u_t \le v_{max} & & (3)\\
 & \sum_{t\in T} v_{it} \le |V| \cdot u_t & \forall i \in V & (4)\\
 & v_{it} + v_{jt} \le 1 & \forall (i,j) \in S, \forall t \in T & (5)\\
 & v_{it} + v_{jt} \le 1 + z_{ijkl} & \forall (i,j,k,l) \in I, \forall t \in T & (6)\\
 & v_{kt} + v_{lt} \le 1 + (1 - z_{ijkl}) & \forall (i,j,k,l) \in I, \forall t \in T & (7)\\
 & v_{it} \in \{0,1\} & \forall i \in V, \forall t \in T & (8)\\
 & z_{ijkl} \in \{0,1\} & \forall (i,j,k,l) \in I & (9)\\
 & u_t \in \{0,1\} & \forall t \in T & (10)
\end{array}
$$

Fig. 3 An integer linear programming solution for a given topological ordering.

We want to minimize the number of different values t that are used. If and only if a t-value is used at least once, u_t is forced to become 1, and we can count these u_t's. Constraint (2) makes sure that each place holder is assigned a tag. Constraints (5), (6), and (7) implement the interference information, where the latter two use a standard formulation for ensuring that one out of two constraints hold. The remaining constraints define domains.

7 Empirical Results

7.1 Different Optimization Criteria

In order to illustrate the influence of the different optimization criteria, we choose a random synthesis tree t with 37 input compounds (leaves) and Strahler number $\mathscr{S}(t) = 4$. The synthesis tree is depicted in Appendix B. According to Section 3, it is always possible to find a DNA-templated program which uses only two different tags on the compounds. This approach will lead to a DNA-templated program with 168 instructions. The program itself, as well as a visualization of all the states of the sequential one-pot synthesis after execution of each of the instructions, can be inspected at `http://cheminf.imada.sdu.dk/dna/`. As $\mathscr{S}(t) = 4$, at least three different strands are needed for the synthesis. As two tags are not enough to create three different strands, a trivial lower bound for the overall number of tags is three.

A naïve approach in order to find *short* DNA-templated programs is to tag each of the input compounds with a different tag. This will obviously lead to no interference and therefore to the shortest possible program. By this approach, we found a program with 109 instructions, using 37 tags. In a practical setting, using unique tags on all compounds likely leads to a more complicated sequence design problem, as there are many more potential but unwanted interferences.

In order to find short DNA-templated programs that use a small number of overall tags, we enumerated all possible depth-first search traversals of the given tree. Each traversal leads to a topological order which in turn is used as input for the ILP approach as presented in Section 6, in order to find an optimal number of tags for the given topological order. Obviously, this approach is only feasible if the given synthesis tree is relatively small, as the number of different depth first search traversals grows exponentially. For the specific example tree, we found a program of optimal length that uses 25 different tags, and no other program with only 109 instructions using fewer tags. Similarly, we optimized with respect to the overall number of tags as a first optimization criterion and the length as a second optimization criterion. This lead to a program of length of 142 using three tags. A summary of these results is given in Table 2.

When optimizing for the overall number of strands, we showed in Section 4 that an optimal result will always be found by recursively completing one subtree before starting the computation for its sibling subtree. We remark that we found examples where this strategy might lead to sub-optimal solutions with respect to the optimal number of overall tags, i.e., it was only possible to find an optimal number of tags by alternating the instructions needed for the synthesis of two sibling subtrees.

7.2 Synthesis Plans with Strahler Number $\mathscr{S}(t) = 6$

In order to empirically analyze the possible sets of strands which can be used in order to perform a specific synthesis successfully, we randomly created approximately 3 million synthesis trees with Strahler number $\mathscr{S}(t) = 6$. In this section, we focus on programs with an optimal number of strands. According to Section 4, it is necessary and sufficient to use 5 strands for trees with $\mathscr{S}(t) = 6$. Note that a set of strands can naturally be represented as a directed graph $G = (V,E)$ without loops. In G, the vertices in V correspond to the set of tags, and the strands correspond to the edges in E, as a strand is a ordered pair of two tags. Of all the synthesis trees, 99.21% could be solved with 3 tags (and 5 strands), which corresponds to the trivial lower bound for the number of tags. The remaining 0.79% of all the synthesis trees (corresponding to 23,780) needed 4 tags (while still using the optimal number of 5 strands). It is easy to infer that there are 37 non-isomorphic directed graphs without self-loops with $|V| = 4$ and $|E| = 5$. Equivalently, there are 37 different strand sets to be considered. Out of those 37 only 13 could be successfully employed in order to solve at least one of the 3 million synthesis plans. The 13 non-isomorphic graphs corresponding to these strand sets are depicted in Fig. 4 and in Appendix C. Only 9 of the 13 strand sets were used in order to solve at least one of the synthesis trees which needed 4 tags. For any choice of one of the 13 strand sets, it could either not at all solve any synthesis trees requiring 4 tags, or it could solve more than approximately 20,000 synthesis trees; details are given in Appendix C. Interestingly, only 2 of the 13 strand sets could be used to solve *all* of the 23,780 synthesis trees that needed 4 tags (depicted with a red border in Fig. 4).

8 Concluding Remarks

We have considered optimizations problems from DNA-templated synthesis. After having developed an appropriate modeling framework, we have focused on automatically inferring DNA-templated programs with various optimization criteria in mind. We have presented optimal and near-optimal methods for minimizing the number of tags and/or the number of strands, and demonstrate how integer linear program-

First OC	Second OC	Algorithm	Tags on Compounds	Tags overall	Strands overall	Length
Tags on Compounds	Tags overall	Section 3	**2**	**3**	**3**	168
Length	—	Naïve	37	37	36	**109**
Length	Tags overall	All traversals + ILP	25	25	36	**109**
Tags	Length	All traversals + ILP	3	**3**	6	142

Table 2 A comparison of inference of DNA-templated programs for the example synthesis tree given in Appendix B according to different optimization criteria (OC); entries in bold signify the optimal values.

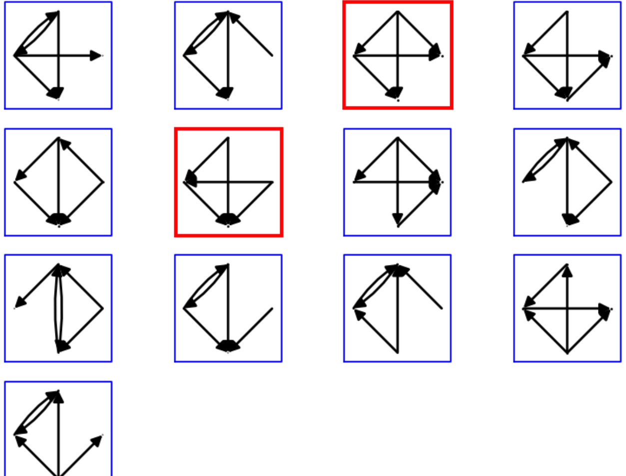


Fig. 4 Shown are the 13 of the 37 non-isomorphic graphs $G = (V, E)$ without self-loops for $|V| = 4$ and $|E| = 5$. These are the only graphs for which the corresponding strand sets could be successfully employed for at least one of a large set of randomly generated synthesis trees. Only two of the 13 strand sets could be used to solve *all* of the 23,780 synthesis trees that needed at least 4 tags (red border).

ming solvers can be employed to consider all possible DNA-templated programs for small problem instances.

Directly related to the questions we consider, it would be interesting to settle the near-optimality issue for complete binary trees, where we have provably optimal results for heights up to 25, except for heights 10–12. It may be necessary to loosen the constraint of using antipaths for the labeling slightly, but it requires great care to still ensure correctness. Also in relation to the complete binary tree algorithm, solutions could be used as the basis for solutions for trees that are not complete. For instance, adding long paths to a complete binary tree need not result in a higher cost in terms of number of tags and strands. It seems that for trees in general, the largest induced complete binary tree is the key to the cost and a formal extension from complete binary trees to trees in general exploiting this kernelization-like idea would be nice.

Based on the empirical evaluation it seems obvious to analyze the different optimization criteria in more detail. For instance, our results show strong empirical support for the hypothesis that the 13 graphs depicted in Fig. 4 correspond exactly to the only strand sets of that size that can be used successfully in the case $\mathscr{S}(t) = 6$. Interestingly, these 13 graphs can also be created in an iterative manner by adding edges to a graph with 4 vertices, where any edge after the first must share either the head or tail of an already existing edge. We hypothesize that successful strand sets can be characterized in such a manner. The length of a inferred program is another important optimization criterion. Besides finding trivial solutions which use a very large number of tags, it would be interesting to find efficient approaches for the automatic inference of Pareto-optimal solutions when considering several optimization criteria.

A quite different direction is to explore concurrency. Using more tags and strands than the bare minimum, some subtrees may become independent and even one-pot synthesis could allow for concurrency. Trade-off results between concurrency maximization and tag/strand set minimization would be interesting.

References

1. L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 5187:1021–1024, 1994.
2. J. L. Andersen, C. Flamm, M. M. Hanczyc, and D. Merkle. Towards optimal DNA-templated computing. *International Journal of Unconventional Computing*, 11(3–4):185–203, 2015.
3. E. Benson, A. Mohammed, J. Gardell, S. Masich, E. Czeizler, P. Orponen, and B. Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523:441–444, 2015.
4. L. Cardelli. Two-domain DNA strand displacement. In *6th Workshop on Developments in Computational Models*, volume 26 of *Electronic Proceedings in Theoretical Computer Science*, pages 47–61, 2010.
5. D. de Werra and C. Pasche. Paths, chains, and antipaths. *Networks*, 19(1):107–115, 1989.
6. A. P. Ershov. On programming of arithmetic operations. *Doklady Akademii Nauk*, 118(3):427–430, 1958.
7. P. Flajolet, J. C. Raoult, and J. Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.
8. R. A. Goodnow Jr, C. E. Dumelin, and A. D. Keefe. DNA-encoded chemistry: enabling the deeper sampling of chemical space. *Nature Reviews Drug Discovery*, 16:131–147, 2017.
9. K. Gorska and N. Winssinger. Reactions templated by nucleic acids: More ways to translate oligonucleotide-based instructions into emerging function. *Angewandte Chemie International Edition*, 52(27):6820–6843, 2013.
10. B. N. Hansen, K. S. Larsen, D. Merkle, and A. Mihalchuk. DNA-Templated Synthesis Optimization. In *23rd International Conference on DNA Computing and Molecular Programming (DNA)*, volume 10467 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2017.

11. B. N. Hansen and A. Mihalchuk. DNA-Templated Computing. Master's thesis, University of Southern Denmark, Denmark, 2015. `http://cheminf.imada.sdu.dk/dna/` [Accessed October 30, 2017].
12. Y. He and D. R. Liu. A sequential strand-displacement strategy enables efficient six-step DNA-templated synthesis. *Journal of the American Chemical Society*, 133(26):9972–9975, 2011.
13. J. B. Hendrickson. Systematic synthesis design. 6. Yield analysis and convergency. *Journal of the American Chemical Society*, 99:5439–5450, 1977.
14. X. Li and D. R. Liu. DNA-templated organic synthesis: Nature's strategy for controlling chemical reactivity applied to synthetic molecules. *Angewandte Chemie International Edition*, 43:4848–4870, 2004.
15. W. Meng, R. A. Muscat, M. L. McKee, P. J. Milnes, A. H. El-Sagheer, J. Bath, B. G. Davis, T. Brown, R. K. O'Reilly, and A. J. Turberfield. An autonomous molecular assembler for programmable chemical synthesis. *Nature Chemistry*, 8:542–548, 2016.
16. I. Nakata. On compiling algorithms for arithmetic expressions. *Communications of the*, 10(8):492–494, 1967.
17. A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 6(Suppl 4):S419–S436, 2009.
18. R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.
19. A. N. Strahler. Hypsometric (area-altitude) analysis of erosional topography. *Bulletin Geological Society of America*, 63:1117–1142, 1952.
20. S. F. J. Wickham, J. Bath, Y. Katsuda, M. Endo, K. Hidaka, H. Sugiyama, and A. J. Turberfield. A DNA-based molecular motor that can navigate a network of tracks. *Nature Nanotechnology*, 7:169–173, 2012.

Appendix A: DNA Program Example

We consider an example synthesis tree with four base compounds. The actual names of the compounds is not used in any of our algorithms, but for illustration, assume the base compounds are A, B, C, and D. Furthermore, we assume that the tagged compound A reacts with the tagged compound B ($A+B \to E$), and that E will have the tag of B. The complete assumptions are

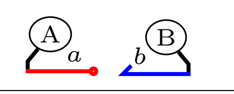
$$\begin{aligned} A+B &\to E, & &E \text{ will inherit the tag of } B \\ C+D &\to F, & &F \text{ will inherit the tag of } C \\ E+F &\to X, & &X \text{ will inherit the tag of } E \end{aligned}$$

and we demonstrate one possible program computing the target compound X as a sequential one-pot synthesis.

We first tag the base compounds A at the left end of the tag a and B at the right end of the tag b. The tag a (respectively b) is depicted as a red (respectively blue) line in the following.

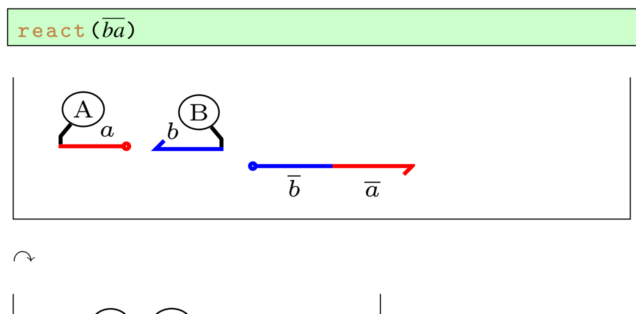
```
1 tag(A, a, left)
2 tag(B, b, right)
```

The state is as follows:



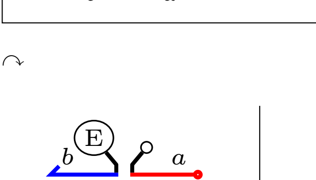
We add the complementary strand $\overline{ba}$ in order to bring A and B in close vicinity and they react to produce E. In this process, A loses its tag.

```
3 react(ba)
```



↷

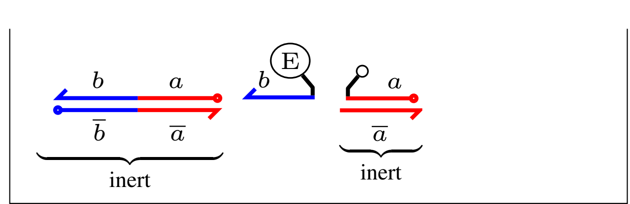
↷



We release the produced tagged compound E with the strand ba and E is now tagged with b. The tag a is now unattached and we add the complementary tag $\bar{a}$ such that in the subsequent operations, it can be ignored.

```
4 release(ba)
```

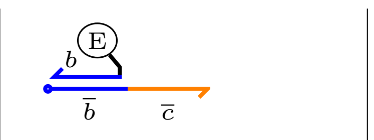
↷



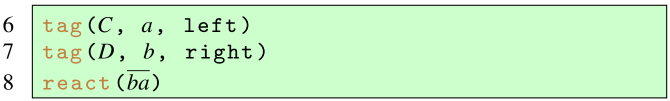
Since they are no longer relevant, we will not depict the inert strands in the following.

In order to avoid unintended interference, we block the tagged compound E with a strand $\overline{bc}$ ($\bar{c}$ shown in orange).

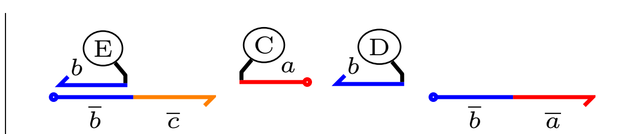
```
5 block(bc)
```



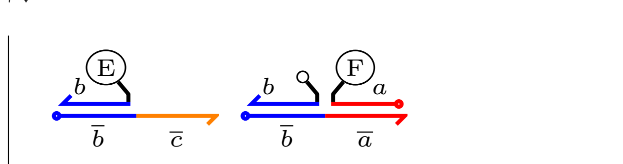
We proceed with the base compounds C and D in a similar manner. Note that C is tagged with a and D is tagged with b, i.e., adding them to the pot in the beginning would have led to unintended interference. By adding $\overline{ba}$, the tagged compounds C and D react to produce F, and D loses its tag.



```
6  tag(C, a, left)
7  tag(D, b, right)
8  react(b̄ā)
```

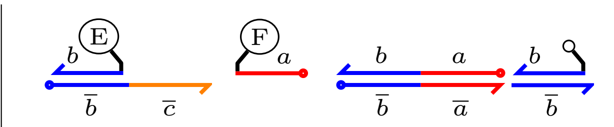


↷



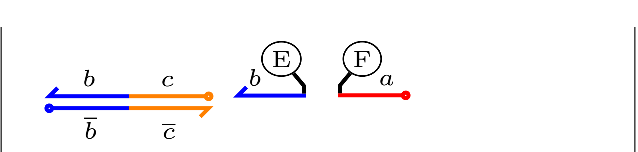
We then release the tagged compound F using the strand ba and pacify the tag b.

```
9  release(ba)
```



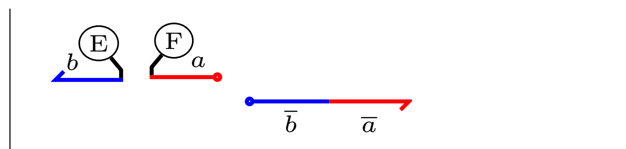
The blocked tagged compound E is released with the strand bc.

```
10  release(bc)
```

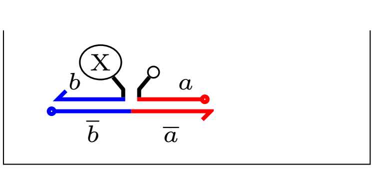


Finally, the tagged compounds E and F are brought in close vicinity using the strand $\overline{ba}$, producing X, and F loses its tag.

```
11  react(b̄ā)
```

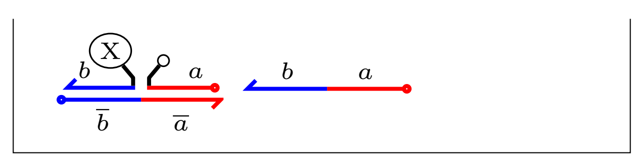


↷

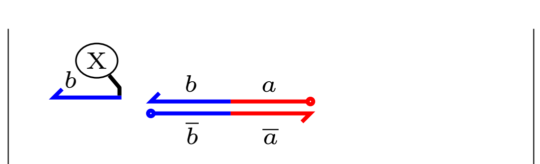


In the very last step, the target compound is released using strand ba, which finalizes the synthesis.

```
12  release(ba)
```



↷



The only non-inert tag is the tag attached to compound X, which makes it chemically easy to extract the compound from the pot. The synthesis required three different tags and two different strands (and their corresponding complementary tags and strands).

The given example also illustrates the minimization of the number of tags for blocking, when assuming that only two tags on the compounds are used (see the definition of MNT) and the number of tags for blocking is to be minimized. Without loss of generality, we choose the goal compound X to be tagged with b. Given that decision, and given that we have restricted ourselves to using only two different tags on the compounds, there are no further choices for tagging: The tagging of all nodes in the tree is simply inferred as follows. The nodes A, C, and F need to be tagged with an a, and B, D, and E with a b. In this example, the subtree of the root X corresponding to $A+B \rightarrow E$ is synthesized before the subtree corresponding to $C+D \rightarrow F$. As we need to block the result of the former synthesis, we need an additional tag for blocking for the subtree E. With respect to the definition of MNT, this corresponds to the recursive calculations for the inference $\max(\textsc{Mnt}(E,0,0), \textsc{Mnt}(F,1,0))$ (the choice to synthesize the subtree $C+D \rightarrow F$ first would, in this specific example, lead to the same overall result). This leads to the following base cases for the leaves: $\textsc{Mnt}(A,0,0) = 0$ and $\textsc{Mnt}(B,0,0) = 0$, and for the other subtree $\textsc{Mnt}(C,1,0) = 1$ and $\textsc{Mnt}(D,1,0) = 1$. Obviously, $\textsc{Mnt}(E,0,0) = 0$ and $\textsc{Mnt}(F,1,0) = 1$, leading to $\textsc{Mnt}(X,0,0) = \min(\max(\textsc{Mnt}(E,0,0), \textsc{Mnt}(F,1,0)), \ldots) = 1$. Thus, only one additional tag is needed for blocking.

Appendix B: Example Tree used for Empirical Evaluation

Figure 5 shows the example tree used for the empirical evaluation.

Appendix C: Details for Empirical Evaluation

In order to perform an empirical evaluation of the possible sets of strands which can be used in order to perform a specific synthesis successfully, we randomly created 2,999,928 synthesis trees with Strahler number $\mathscr{S}(t) = 6$, using the following recursive process: If the Strahler number s does not correspond to just one node (a leaf), then we create a node and generate its subtrees as follows. With probability $2/3$, we recursively generate two subtrees, both of which have Strahler numbers $s-1$. With probability $1/3$, we let one subtree have Strahler number s and choose uniformly at random between the Strahler numbers one through $s-1$ for the other subtree. In all cases, the ordering of the subtrees (left or right) are decided upon uniformly at random.

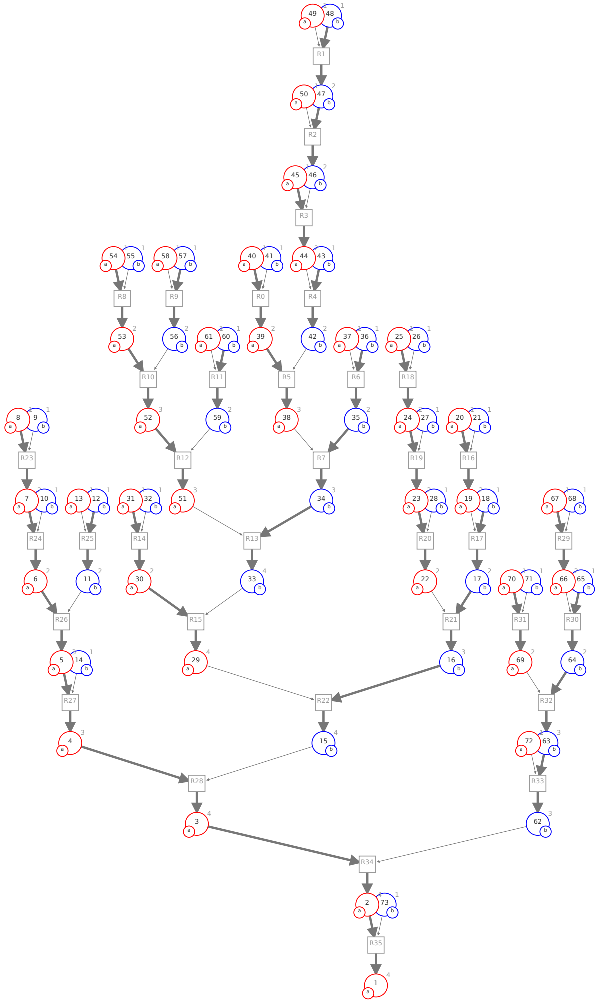


Fig. 5 Example synthesis tree used for the empirical evaluation with different optimization criteria as also visualized at `http://cheminf.imada.sdu.dk/dna/`. The coloring of the nodes is chosen according to the optimization of the number of tags on compounds, bold edges indicate tag inheritance, the left and right end tagging is indicated by the small circles to the left and right, respectively, of the large circles, which represent the compounds. The DNA-templated program, which uses 2 tags on the compounds (here red and blue) and 3 tags overall, has a length of 168 instructions. The DNA-templated program itself, a visualization of the sequence of state changes for the sequential one-pot synthesis, as well as statistical information can also be found via the before-mentioned URL. The shortest possible DNA-templated program, which can easily be found by tagging all 37 input compounds with a different tag, has a length of 109 instructions. By an exhaustive enumeration of all tree traversals and employing the ILP-based brute force approach for each of the traversals, a program of length 109 was found that uses the minimum 25 tags for that program length.

Strand set	k
$\boldsymbol{\{ab,ac,ad,bc,bd\}}$	**23780**
$\{ab,ac,ad,bd,cd\}$	23769
$\{ab,ac,ad,bd,db\}$	22161
$\{ab,ac,ad,cd,db\}$	22161
$\boldsymbol{\{ac,ad,bc,bd,cd\}}$	**23780**
$\{ac,ad,bd,ca,cd\}$	22106
$\{ac,ad,bd,cd,dc\}$	21874
$\{ac,ad,ca,cb,cd\}$	21796
$\{ad,bc,bd,ca,cd\}$	22106
$\{ac,ad,bd,da,dc\}$	0
$\{ac,ad,bd,db,dc\}$	0
$\{ac,ad,cd,db,dc\}$	0
$\{bc,bd,ca,cd,da\}$	0

Table 3 Shown are the 13 out of the 37 possible strand sets with 4 tags, that could be used for at least one of the 2,999,928 synthesis plans. k out of the 23,780 synthesis plans were solved with the corresponding strand set.

Of the 37 possible strand sets which use 4 tags, only 13 were able to solve at least one synthesis plan. 23,780 of all the synthesis plans could not be solved with a strand set based on 3 tags (and 5 strands), but required 4 tags (and 5 strands). Only 9 of the 13 strand sets could be used to solve at least one of the 23,780 synthesis plans and 2 of the 9 strand sets could be used for all 23,780 synthesis plans; see Fig. 4 and the details given in Table 3.