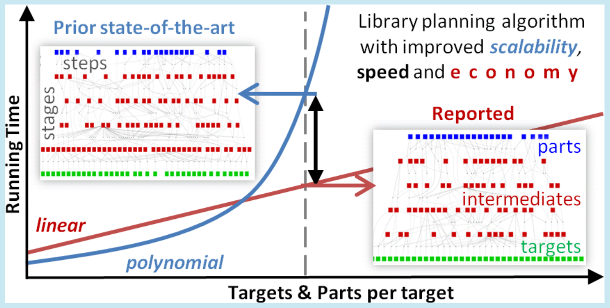
Heuristic for Maximizing DNA Reuse in Synthetic DNA Library Assembly

Jonathan Blakes,[†] Ofir Raz,[†] Uriel Feige, Jaume Bacardit, Paweł Widera, Tuval Ben-Yehezkel, Ehud Shapiro,* and Natalio Krasnogor*

Supporting Information

ABSTRACT: *De novo* DNA synthesis is in need of new ideas for increasing production rate and reducing cost. DNA reuse in combinatorial library construction is one such idea. Here, we describe an algorithm for planning multistage assembly of DNA libraries with shared intermediates that greedily attempts to maximize DNA reuse, and show both theoretically and empirically that it runs in linear time. We compare solution quality and algorithmic performance to the best results reported for computing DNA assembly graphs, finding that our algorithm achieves solutions of equivalent quality but with dramatically shorter running times and substantially improved scalability. We also show that the related computational problem *bounded-depth min-cost string production* (BDMSP), which captures DNA library assembly operations with a simplified cost model, is NP-hard and APX-hard by reduction from vertex cover. The algorithm presented here provides solutions of near-minimal stages and thanks to almost instantaneous planning of DNA libraries it can be used as a metric of "manufacturability" to guide DNA library design. Rapid planning remains applicable even for DNA library sizes vastly exceeding today's biochemical assembly methods, future-proofing our method.


Prior state-of-the-art
steps
stages
Running Time
Library planning algorithm with improved scalability, speed and economy
Reported
parts
intermediates
targets
linear
polynomial
Targets & Parts per target

KEYWORDS: *DNA libraries, library assembly algorithm, synthetic biology, computational hardness*

The ongoing next-generation DNA sequencing revolution has been driven during the past decade by the advent of several disruptive technologies, which have converged to both reduce the cost[1] and increase the throughput[2] of DNA sequencing by a factor of 10^5 compared to traditional Sanger sequencing. In contrast, current technology for *de novo* DNA synthesis is trailing far behind sequencing by a factor of 10^6 per base in terms of cost and by an even larger factor in terms of throughput.[3] It is generally acknowledged that new disruptive technologies for DNA processing are needed for advancing the development of biotechnology and biomedical research.

In concert with entirely *de novo* synthesis, a swathe of alternative DNA assembly methods are being introduced[4−6] that concatenate parts from hundreds of base pairs in length, through gene fusions, to synthetic biological devices built from catalogued parts, up to megabase fragments and entire chromosomes.

Both combinatorial DNA assembly and *de novo* DNA synthesis can be improved considerably (e.g., made cheaper, ameliorate error rates, etc.) by maximizing DNA reuse—identifying shared parts and utilizing them during construction. Most DNA generally needed in biological and biomedical research, and in synthetic biology and systems biology in particular, are extensions, variations, and combinations of existing DNA. This provides ample opportunities for DNA reuse. *De novo* synthesis relies on synthetic oligos that are inherently error prone, and therefore, reusing existing error-free DNA in constructing new DNA provides an inherent advantage. Moreover, even in applications that require completely new DNA, when adapting codon usage,[7−9] for example, an exploratory library of variants is typically needed.[10] The production of these variants, once an error-free prototypical instance of the DNA molecule is isolated or synthesized, includes opportunities for DNA reuse almost by definition.

Rigid DNA production methods that follow a fixed production plan are inherently prevented from incorporating DNA reuse. Achieving reuse requires sophisticated DNA construction optimization algorithms, as well as flexible automated production systems[11,12] that can take advantage of construction plans produced by such algorithms.

In this paper, we show how combinatorial DNA libraries (graphically represented in Figure 1) can be assembled algorithmically in an efficient manner from pre-existing and newly synthesized biological parts. This is achieved by recognizing that consecutive combinations of input parts appear multiple times within the output molecules (targets). Ordering the necessary assembly steps into stages, in which a judicious selection of intermediates are assembled earlier, facilitates reuse of assembled parts, leading to an overall reduction in the number of biochemical operations required to assemble the library. A fast heuristic for library planning and DNA reuse, such as the algorithm we propose in this paper, is an essential component in the algorithmic arsenal of Synthetic Biology.

Special Issue: IWBDA 2013

Received: October 8, 2013
Published: February 20, 2014

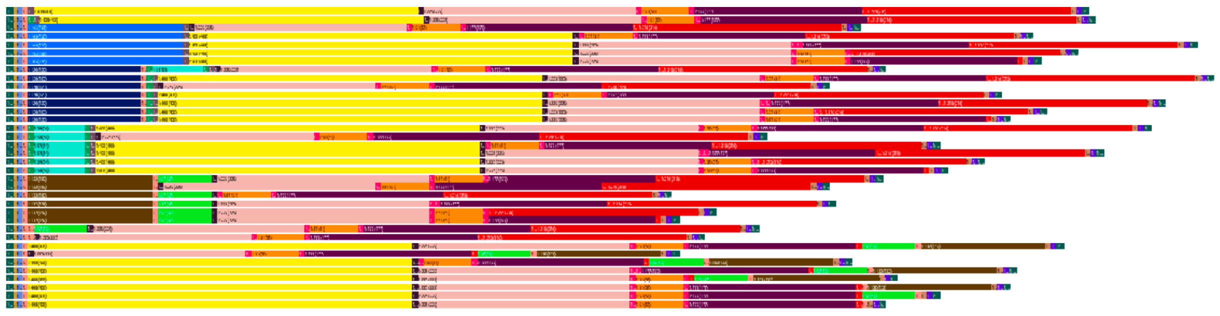


Figure 1. Composition of a DNA library with 36 targets, stacked to show relative lengths and the distribution of the parts; different colors represent reusable parts from different provenances. Note that some successions of parts are shared between several targets. Assembly graphs for this library, of the kind shown in Figure 2, can be found in Figure 4.

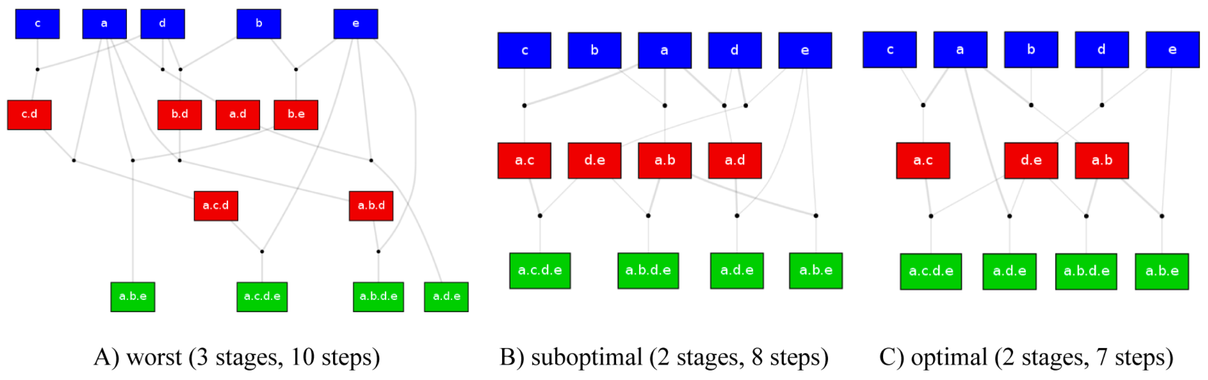

c
a
d
b
e
c.d
b.d
a.d
b.e
a.c.d
a.b.d
a.b.e
a.c.d.e
a.b.d.e
a.d.e
A) worst (3 stages, 10 steps)
a.c
d.e
a.b
a.d
B) suboptimal (2 stages, 8 steps)
C) optimal (2 stages, 7 steps)

Figure 2. Graphs of three alternative valid assembly plans for set of four sequences {*acde, abde, ade, abe*}. Primitive parts are shown in blue, intermediate composites in red, and targets in green. From the top downward, each step is represented by a black dot concatenating two parts from an earlier stage into a new intermediate or target part. The set of vertically aligned steps forms a stage. The thicker line from a part to a dot/step indicates the left component of the concatenation.

This is so due to the fact that, as DNA synthesis becomes cheaper, very large combinatorial DNA libraries will become more prevalent and thus efficient tools for handling these ever increasing DNA libraries are needed.

A scenario where large combinatorial DNA libraries are needed include extant Synthetic Biology approaches for the efficient production of fine chemicals or other commodity products (e.g., antibiotics) that require the manipulation and fine-tuning of various enzymes and their coordination into a set of one or more synthetic pathway. The engineering of the latter must be done in a way that preserves, as close as possible, the physiological viability of host cellular factories. A key determinant of viability and efficient production is the concentration the various enzymes adopt in the engineered pathways. In turn, these concentrations can be controlled by, for example, modifying ribosome binding sites, promoters' strengths, degradation tags, etc. Combinatorial DNA libraries capturing a large variants set of a synthetic pathway are but one example of the need for efficient planning algorithms for synthetic DNA library assembly.

A DNA library assembly task includes a set of input molecules and a set of output molecules (targets) and assumes a binary part concatenation operation that can select two parts within existing (input or intermediate) molecules to produce their concatenation. An assembly plan that produces the outputs from the inputs can be viewed as a graph representing the binary part concatenation operations leading to the set of targets from the set of inputs through a number of intermediates. Figure 2 visualizes three possible plans for constructing a simple library composed of four targets. The quality of a plan can be categorized by the pair of integers (*stages*, *steps*), which allows to compare the outputs of assembly planning algorithms. A plan is considered to be better if it has fewer stages and fewer steps. There may be a trade-off between stages and steps; however, whether a suboptimal number of stages with significantly fewer steps is preferable or not is a technology-dependent manufacturing decision.

More formally, we will refer to strings over a given fixed finite alphabet (i.e., of nucleotides). The part concatenation operation applied to a set of strings $\boldsymbol{X}$ consists of selecting two strings A and B from $\boldsymbol{X}$, selecting a part (consecutive substring) A' of A and a part B' of B and adding the string A' followed by B', denoted $A'B'$ back to $\boldsymbol{X}$.

An *S-T library assembly problem* is to produce the set of target strings $\boldsymbol{T}$ using a finite number of applications of the part concatenation operation on $\boldsymbol{S}$.

We note that the part concatenation operation is realizable in most cases using standard biochemistry. Part selection can be achieved via PCR amplification. Concatenation can be realized using various binary assembly protocols.[4] The cost of a solution is defined by the number of concatenation operations. The depth of a solution is defined as the minimum number of parallel stages needed, where a parallel stage is a set of concatenation operations that can be performed in parallel since they do not depend on each other. Equivalently, the depth of a solution is the maximum number of concatenation steps on a directed path from a source string to a target string. We also consider the problem of finding a solution of depth at most d and minimum cost and call this problem *bounded-depth min-cost string production* (BDMSP).

```
 1  T′ ← set of targets designed with available parts in S
 2      as set of m-tuples of n-tuples e.g. {((a,b),(c)), ((d),(e,f)), ((a,b),(c),(d),(e,f))}
 3  stages ← empty list to populate with planned steps in run (each stage is a set of steps)
 4
 5  function plan_assembly_stages_iterative(T′, stages)
 6      loop until return on line 37
 7          1. count overlapping pairs T′ and identify trios in single pass
 8          pairs ← empty counter of 2-tuples of n-tuples      e.g. {((a,b),(c)) : 1}
 9          trios ← empty set of 3-tuples of n-tuples          e.g.  ((a,b),(c),(d,e,f))
10          for each parts sequence s in T′ do
11              for each index i in [1st..penultimate] do
12                  add (s[i],s[i+1]) to pairs
13                  if i < penultimate then
14                      add (s[i],s[i+1],s[i+2]) to trios
15
16          2. map pairs to containing trios (and vice versa)
17          pair2trios ← empty map of pairs to sets of trios
18          trio2pairs ← empty map of trios to sets of pairs
19          for each trio tr in trios do
20              lp ← (tr[1st],tr[2nd]); rp ← (tr[2nd],tr[3rd])  recreate left and right pairs
21              add tr to pair2trios[lp] and pair2trios[rp]          use pairs to lookup trios
22              add lp to trio2pairs[tr] ; add rp to trio2pairs[tr]   use trio to lookup pairs
23
24          3. shuffle then stably sort pairs to reorder within rankings
25          pair_counts ← list of (pair,count) 2-tuples from pairs counter
26          in-place shuffle pair_counts
27          in-place sort pair_counts stably by count descending
28
29          4. add non-excluded pairs to stage and exclude overlapping using maps from 2
30          excluded ← empty set
31          for each pair p in pair_counts do
32              if p not in excluded then
33                  for each tr in pair2trios[p] do
34                      add all overlapping trio2pairs[tr] to excluded
35                  add p to stage
36
37          if 0 steps in stage then return stages else add stage to stages
38
39          5. simulate assembly stage updating T′ sequences in-place
40          for each pair p in stage do
41              L ← p[1st]; R ← p[2nd] e.g., ((L1, L2, L3), (R1, R2))
42              p′ ← concatenation of parts in L then R an l+r-tuple, e.g., (L1, L2, L3, R1, R2)
43              for each sequence s in T′ do
44                  for each starting index i of an occurrence of p in s do
45                      replace s[i] with p′ and s[i+1] with () the empty 0-tuple
46                      to preserve indices; and avoid, e.g., s = ((a),(a),(a)) and p' = (a,a)
47          for each sequence s in T′ do
48              excise 0-tuples from s
49
50          6. loop to populate stages
```

Listing 1. Pseudo-code for new library assembly algorithm.

The present state-of-the-art heuristic for BDMSP is the multiple goal-part assembly algorithm,[13] which iteratively assembles individual targets using the minimum number of stages, producing a high quality library assembly graphs using three techniques: (1) making accumulated subgraphs available to subsequent targets at no cost, (2) biasing subgraph choice based on precomputed counts of all possible intermediates, and (3) permitting the depth of any target tree to grow up to the global minimum depth (‘slack’). The multiple goal-part assembly algorithm was shown analytically to have a runtime complexity $O(k^2 n_{\max}{}^3)$, where k is the number of targets and $n_{\max}$ the number of component parts (n) in the largest target; a result supported by small scale empirical evidence.

To validate their algorithm, the authors use two real-world libraries and one small artificially generated one. They showed that the algorithm had lower costs, that is, it used far fewer *steps*, than the best solutions produced by random search. Also, that at least for the small ($k = 5$, $n_{\max} = 5$) artificial library, the optimal solution, as determined by exhaustive search, could be found.

Problem Solved? It would be desirable, for example, in CAD software that enables synthetic biologists to design libraries of DNA-based devices in a combinatorial manner, to have an *accurate* measure of the optimal number of biochemical operations required to assemble a given library. An integrated decision-support mechanism leveraging that measure could guide the researcher in making the most effective use of limited procurement and analytical resources. Knowing the value of the near optimal *stages* and *steps* objective function of solution quality would be an effective means by which to benchmark and differentiate algorithms addressing this problem.

When assembly steps are restricted to binary concatenations, the minimum number of stages can be calculated[13] as $ceil(\log_2(n_{\max}))$. Determining the minimum number of steps is more difficult. In what amounts to solving the problem, the minimum number of steps possible *could* be determined by exhaustively enumerating all possible assembly plans, but for even modestly sized libraries this is an intractable task: for just a single target the number of possible assembly graphs is the Catalan number:[13] $C_n = (2n)!/(n+1)!n!$, which grows very rapidly with target length; e.g., $C_{[1..5]} = (1, 12, 360, 20160, 1814400)$. Furthermore, we show here that the bounded multistage DNA library assembly problem (BDMSP) is in fact NP-hard[14] and APX-hard,[15] unless $P = NP$, indicating that to determine the minimum number of steps for minimal stages, and even to approximate it up to a constant error, requires a

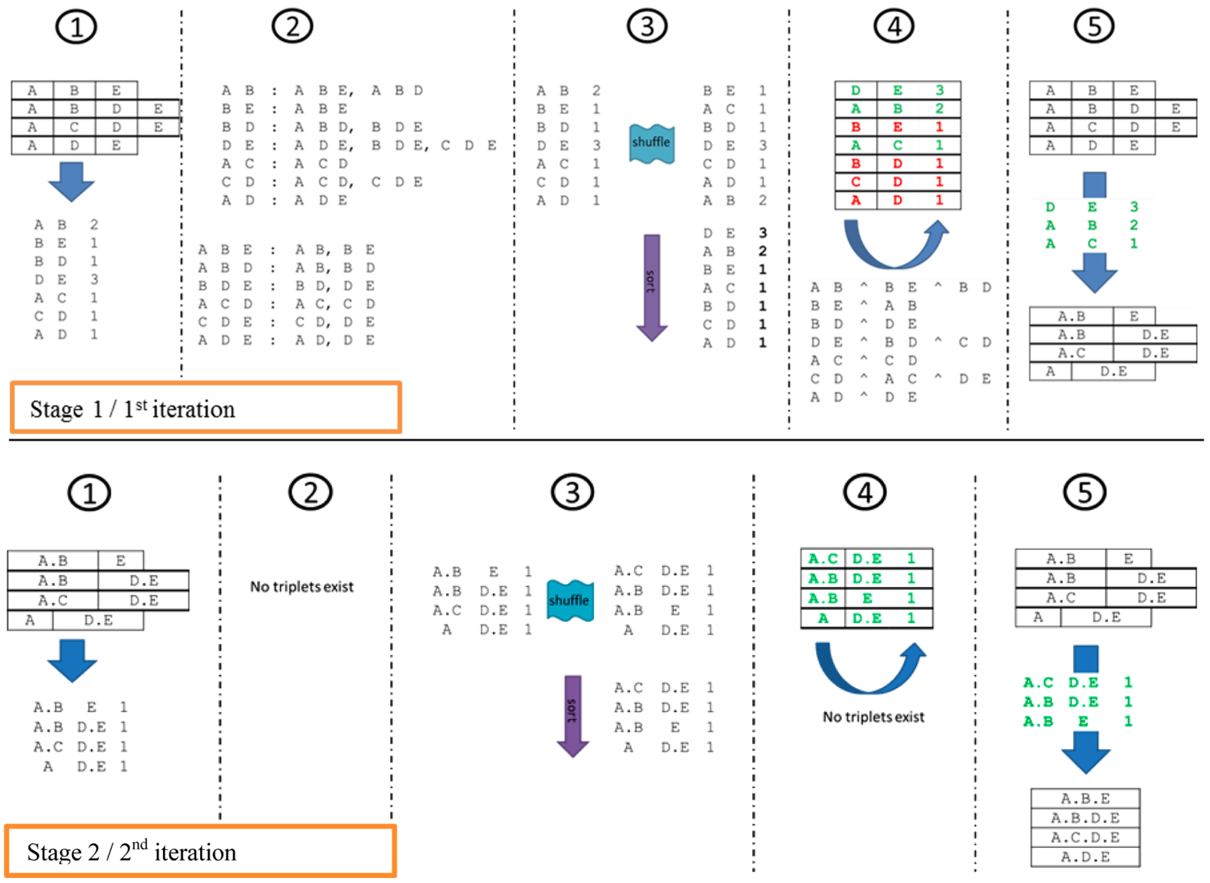


Figure 3. Step-by-step walkthrough of a run with the example library {*ABE, ABDE, ACDE, ADE*} from Figure 2. While targets remain to be assembled, each iteration progresses in five phases: (1) occurrences of every pair are counted; (2) pairs are mapped to containing triplets and vice versa; (3) a list of pair-count tuples is shuffled (blue wave) and stably sorted by count descending (purple arrow); (4) for each yet-to-be-excluded pair, choose pair to be a step (green rows) then exclude other pairs (red rows) sharing the same containing triplets using mappings from the second phase; finally (5) concatenate all instances of each step-pair. Note that in this example, no triplets exist in the second iteration, as the structure of the library after the first iteration is exactly the set of pairs that will comprise the steps of this final stage.

superpolynomial number of operations. In these circumstances, robust (fast and scalable) assembly algorithms that do not sacrifice solution quality for performance are required.

RESULTS AND DISCUSSION

Computational Hardness of BDMSP. A minimization problem Π is said to be APX-hard if there is some constant $\delta < 1$ such that achieving an approximation ratio of δ on Π is NP-hard. In particular, it is known that minimum vertex cover (finding the smallest set of vertices that touches every edge in a graph) is APX-hard, and this holds even on graphs of degree at most 3.

Theorem: *BDMSP is NP-hard and APX-hard. Furthermore, this holds even when S is composed only of single characters and* $d = 2$, *and there is a total order among characters such that in every string of T this order is respected.*

Proof: By a reduction from vertex cover (VC). Full details are provided in Appendix I.

The theorem above applies to bounded-depth min-cost string production (BDMSP), whereas the problem addressed with our new algorithm is that of min-cost string production (MSP) with no depth restriction. We address this closely related problem as it lends itself to a simple and experimentally efficient heuristic. It would be desirable to determine whether MSP is NP-hard as well, but our current work does not answer this question.

New Greedy Algorithm for DNA Library Assembly. We present a new algorithm that directly addresses whole library assembly by operating, at each stage, on all potential binary concatenations without *a priori* depth constraints. The algorithm assembles all targets simultaneously from their individual parts, akin to the visualization of an assembly graph (as can be seen in Figures 2 and 4, and Figure 12B in Appendix I). The central idea is recognizing that maximal reuse of an intermediate part can result in fewer steps (though not necessarily stages) by removing the need for additional steps to reach a given target (Figure 2B and C). This is achieved by greedily concatenating the most frequently occurring pairs of adjacent parts, in the hope of producing only those members of the smallest intermediate parts set. It is well-known that greedy algorithms, applying locally optimal choices at each stage, can quickly obtain solutions that approximate the global optimum, even for some hard problems.

Listing 1 gives detailed pseudocode for the general algorithm. A Python implementation can be found in the Supporting Information. Listing 2 in Methods uses a higher-level, recursive pseudocode for runtime complexity analysis.

Starting from a data structure where each library target is decomposed into a sequence of available unitary parts, the algorithm repeatedly—in analogy to independent steps within an assembly stage—concatenates a subset of the remaining pairs of adjacent parts within the sequences, until no pairs remain, meaning every target has been successfully reassembled. The sets of pairs concatenated at each stage of the execution thereby constitute a viable assembly plan.

Three heuristics operate to influence the choice of assembly steps, for each stage:

(1) Each class of pair is treated as being mutually exclusive to another if at least one instance of each overlaps: shares a right or left part respectively such as *B* in the pairs *AB* and *BC*, of the trio *ABC*. This makes it possible to safely concatenate all instances of any given pair, because instances of overlapping pairs will be prevented from being simultaneously assembled in the same stage. For greedy *efficiency*, all nonexcluded pairs are concatenated.

(2) For *efficacy*, the choice of which pairs to concatenate is determined by the relative number of times each pair occurs within the nascent library data structure. Pairs are sorted by count, which establishes a ranking, but are then processed individually and either chosen as a step or excluded. The first few pairs to be processed are much more likely to be chosen because they are less likely to have been excluded by a previous pair. Those with lower or equal frequencies, assessed later, are likely to have already been excluded and cannot therefore be chosen at that stage.

(3) To enable the broadest exploration of the space of assembly plans obtainable under the previous heuristics, we introduce an element of nondeterminism to every stage. A random shuffle followed by a stable sort serves to reorder pairs with the same count while retaining the ranking, such that the most frequently occurring pairs are still selected and will exclude less frequent ones, dependent only the composition of targets and the library's amenability to assembly by binary concatenation, as captured by pair-counting. This combats the potential for the introduction of bias toward the first encountered pairs, as due to the counting operation necessarily iterating over and within the targets, the ordering of the targets and the position of the pairs along them could, depending on implementation, determine the order in which the pair-counts are stored and the pairs presented for selection and mutual exclusion.

When combined, these heuristics serve to prioritise the production and maximize the reuse of intermediate parts with the highest potential for reuse throughout the library. The most frequent pairs will always exist within the most frequent trios and so on. This results in assembly plans with almost as few as possible stages and steps. For example, in assembling the single target *ABCABC* given the parts {*A*, *B*, *C*}, the algorithm will always produce two instances of either *AB* or *BC* in the first stage. In the second it must produce two instances of *ABC* in preference to only one of *CAB* or *BCA*. *ABCABC* is the only possible concatenation product left, and so, the algorithm terminates having planned the target assembly in the minimum of three stages and three steps. Figure 3 gives another worked example, yielding the solution in Figure 2C.

Computational Validation. In evaluating assembly algorithm efficacy, we use (*stages*, *steps*) as the biobjective function for quantitatively comparing the quality of assembly plans introduced earlier. For multiple runs, we take the mode (or modal average), as it is the most appropriate statistic for the categorical (*stages*, *steps*) tuple. We use the ″best″ and ″worst″ values from multiple runs to indicate the range of solutions found by an algorithm on a particular data set. Efficiency is measured as the mean running time in seconds that a library assembly algorithm requires to compute a valid assembly plan—intended to facilitate simple calculation of potential running times for multiple runs and the number of runs that are achievable in a given time period.

In what follows, we have called our new algorithm A_{new} and the multiple goal-part assembly algorithm A_{mgp}. Results previously obtained for A_{mgp} might have provided the most direct means by which to assess whether A_{new} is an improvement over the state-of-the-art, but only a handful of *steps* values could be extracted from the published figures.[13] We coded $\mathbf{A_1}$ to be a faithful implementation of A_{mgp}, following the available pseudocode and literature, which enabled us to gather statistically significant samples of efficacy measurements, and conduct a fair comparison of efficiency by running both the new and current state-of-the-art algorithm on the same technology stack.

When implementing A_1, we discovered that while A_{mgp} appears to be deterministic—consistently returning the same solution for the same list of targets (an assumption perpetuated by the solitary data points in the original publication)—that perceived determinism only holds for a given ordering of the target list—a consequence of always preferring the first of two equal cost subgraphs—particularly when previous iterations have reduced the cost of already used subgraphs to zero. By shuffling the targets list between repeated runs of A_1 on the same set of targets, we were able to get a broader indication of the solution space explored by A_{mgp} and obtain distributions of solutions that enabled the more nuanced comparison of A_{new} and A_{mgp} efficacies for any given library.

By design A_{mgp}, and therefore A_1, do not return solutions with greater than minimal stages, whereas A_{new} has no means of controlling the number of stages in an assembly, as our results demonstrate. The depth of an A_{new} assembly graph is a function of the target compositions, the heuristics employed and the element of chance. The number of stages in an A_1 produced assembly graph can be increased through the *extra_slack* parameter, which we use to enable a *steps*-only comparison of A_1 and A_{new} with equal stages, when A_{new} can only yield plans with nonminimal *stages*, in Figure 6.

For an approximation of a random search baseline algorithm, using A_{new}, we can simply randomize the choice of steps by disabling the stable sort that succeeds the per-stage shuffle (step 3 in Figure 3). We call this algorithm $\mathbf{A_{new}^{rand}}$ and use it to demonstrate the benefits of prioritizing more common pairs, when greedily applied, over random choice (this is analogous to how the authors of A_{mgp} assessed improvement over random pair concatenation, in the absence of another library assembly algorithm to compare to).

We evaluated A_{new}, against the other algorithms described above, using real-world combinatorial DNA libraries and samples of tangentially related targets from biological parts/device repositories. The identity and characteristics of these benchmarks are summarized in Table 3 of the Methods, Data Set Curation subsection.

Efficacy Evaluation. The ranking heuristic of A_{new} prioritizes pairs of parts (potential concatenation steps in the current stage) that occur most frequently over all library targets, including pairs within the same target. The Fisher–Yates shuffle[16] used in our implementation of A_{new} performs an unbiased randomization of the order in which pairs are considered as potential concatenation steps, so that the sort is absolutely necessary to obtain the required rank ordering. To demonstrate

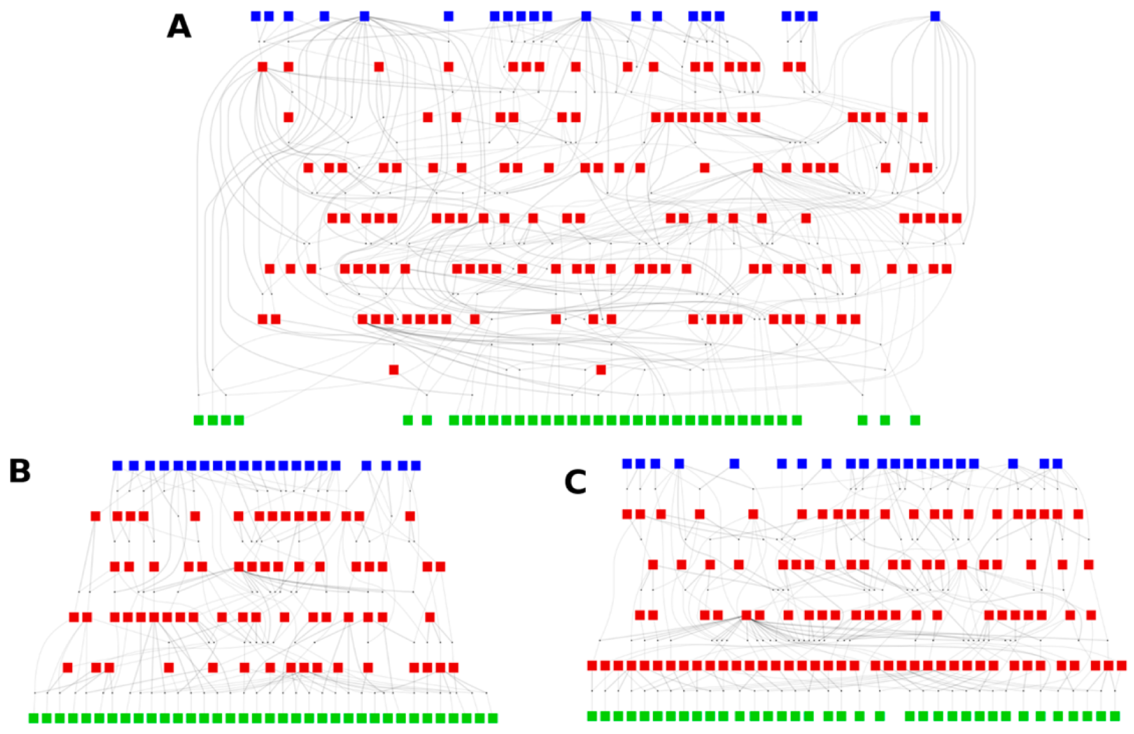


Figure 4. Representative mode quality assembly graphs for UKB library of (A) A_{new}^{rand} (8, 184), (B) A_{new} (5, 102), and (C) A_1 (5, 139).

the importance of this to A_{new} obtaining good quality assembly plans, we compared the assembly graphs generated by A_{new} and A_{new}^{rand} in which the rank-inducing stable sort has been disabled.

Figure 4 shows assembly graphs of the relatively small UKB library (visualized in Figure 1) that are representative of mode quality solutions from 1000 runs each of A_{new}^{rand}, A_{new}, and A_1. The A_{new}^{rand} plan in Figure 4A has the most (8) stages and many more intermediates (red squares); resulting from the concatenation of infrequently occurring pairs that would normally be excluded by earlier choice of other higher ranking pairs. Although some intermediates are quite highly reused, as can be seen from the large number of edges extending downward from some vertices, excess paths indicate non-cooperative routes to the assembly of targets resulting from generally poor intermediate choices. The more efficacious plans seen in Figure 4B and C, while similar in size and shape to each other, allow a distinction to be made between those produced by A_{new} and A_1: the most highly reused intermediate is generated in the second and third stage, respectively. This reflects the greedy choice made at every decision point by $A_{new,}$ for the most likely candidate for a reusable intermediate.

The impact of ranking in the improvement over random is shown even more starkly in Figure 5, where 200 000 runs each of A_{new} and A_{new}^{rand} on the larger iGEM-2008 library were performed (taking ~45 min per algorithm to complete) and the results plotted as distributions of *steps* grouped by the number of *stages*. The green peaks of A_{new} are tightly defined, indicating convergence to a small space of 5 and 6 stage solutions, around 825 steps, where it is likely that the same solutions were repeatedly obtained. As expected for a random search, the red peaks of A_{new}^{rand} show a less-focused, Gaussian-distributed exploration of the solution space, in terms of both steps (~1250) and stages (predominantly 6 or 7; very few 5 stage solutions compared to A_{new}). From this experiment, we

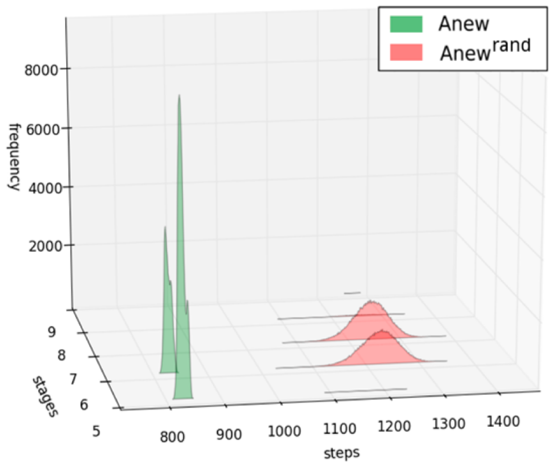


Figure 5. Distribution of solution qualities (200 000 each) for A_{new} (green) and A_{new}^{rand} (red). *Stages* and *steps* are plotted on separate axes to indicate the proportions of solutions with different numbers of stages, that is, for $A_{new,}$ solutions with 5 stages were more common than those with 6 stages.

conclude that the ranking component of A_{new} ensures that considerably fewer steps are needed, and consequently, the number of stages is reduced also.

It is important to note that no solutions with 4 stages, the minimum for iGEM-2008, were found in any of the 400 000 runs summarized in Figure 5. The extra stages required by A_{new} here are evidence that sometimes the greedy heuristic is incapable of achieving minimal stages for certain sets of targets. This is easily proven: given the basic parts {*A*, *B*, *C*, *D*, *E*} and the targets {*ABCD*, *BCE*}, the minimal number of stages is 2, yet the greedy strategy will always concatenate *B* and *C* into *BC* in the first stage, as it is the only pair that occurs more than once. Thereafter, a further 2 stages of binary concatenations are always required to assemble the targets, making a total of 3 stages. Conversely, the same heuristic, problematic for iGEM-2008, can still produce solutions with minimal stages for other instances, such as the Phagemid library shown in Tables 1 and 2.

Table 1. Bi-objective Function Values (*Stages, Steps*) by Algorithm for Each Library Used to Evaluate A_{mgp}

library	statistic	A_{mgp}	A_1	A_{new}
small artificial (k = 5, n_{max} = 5)	best	(3, 11)	(3, 11)	(3, 11)
	mode		**(3, 11)**	**(3, 11)**
	worst		(3, 11)	(3, 15)
Phagemid (k = 131, n_{max} = 10)	best	(4, 205)	(4, 204)	(4, 202)[a]
	mode		**(4, 207)**	**(4, 202)**
	worst		(4, 236)	(4, 208)[a]
iGEM-2008 (k = 395, n_{max} = 14)	best	(4, 820)	(4, 819)	(5, 810)/(6, 807)[b]
	mode		**(4, 835)**	**(5, 824)**
	worst		(4, 855)	(5, 842)/(6, 841)[b]

[a]Solutions of Phagemid by A_{new} with values other than (4, 202) or (4, 208) were never observed. [b]Best and worst solutions that were found with more stages but fewer steps. (Not applicable for A_1 where *stages* is fixed, or mode values.)

Table 1 shows for 1000 runs of A_{mgp}, A_1 (our implementation of A_{mgp}) and A_{new}, the best, mode and worst values of the (*stages, steps*) biobjective function, for each of the libraries originally used to validate A_{mgp}. In each case the results produced by A_1 show strong agreement with the published values for A_{mgp}.[13] Therefore, we can conclude that our A_1 implementation is correct and a suitable proxy for A_{mgp} going forward.

The fourth and fifth columns of Table 1 compare the efficacies of A_1 and A_{new}. For the small artificial library, the mode value of both algorithms agrees and is also the best. The worst value demonstrates that A_{new} will occasionally yield solutions with excess steps resulting from specific combinations of target compositions and the induced orderings.

For both the Phagemid and iGEM-2008 libraries, A_{new} consistently yielded assembly plans with fewer steps than A_1. For iGEM-2008, however, A_{new} always required at least one and at most two more than the minimum of 4 stages. Interestingly, the best and worst plans with 6 stages had a few fewer steps than those their equivalents with 5 stages, possibly indicative of a pareto-front in the design space of combinatorial DNA library assembly plans or simply an artifact of the iGEM-2008 data set.

To investigate the relationship between additional stages and fewer steps and for direct comparison of A_1 solutions with the same number of stages as A_{new}, we used the *extra_slack* parameter of A_1 to increase the depth (*stages*) to which assembly graphs for individual targets in A_{mgp} are allowed to grow. 1000 runs each of A_1 with 1 and 2 extra slack were performed on iGEM-2008 and the resulting distributions plotted in Figure 6 together with the results of A_{new} and A_1 summarized in Table 1. Each set of 1000 A_1 runs required roughly 14 h of CPU time, while 1000 runs of A_{new} took just 14 s.

Figure 6 shows that compared to A_1 with 4 stages (0 extra slack, purple), the right tail of the steps distribution for A_1 with 5 stages (1 extra slack, cyan) is shifted to the left and the mode too slightly, but the left tail (solutions with the fewest steps) is the same at ~820. However, with 2 extra slack (6 stages, green), the distribution narrows considerably and is generally shifted toward fewer steps, with a best of ~815. The distribution of steps for A_{new} (red) with 5 stages is similar in shape to, but shifted left of, A_1 with 1 extra slack (cyan). The steps distribution for A_{new} with 6 stages is wider than for A_1 with 2 extra slack (green), but its best and mode values are found to the left of the latter's mode. We cannot compare the distribution of A_{new} solutions to those of A_1 without extra slack for iGEM-2008, as there are no examples for A_{new} with 4 stages.

To statistically infer whether A_{new} produces solutions with significantly fewer steps than A_1 for the same number of stages, we tested the null hypothesis that pairs of steps distributions *with equal stages* (Figure 6, overlapping) are the same, against the alternative hypothesis that the respective A_1 steps distribution has larger values than that of A_{new}, using the Mann−Whitney-Wilcoxon rank sum test (also called the Mann−Whitney U test). The steps distribution of A_{new} solutions with 5 stages has a significantly different distribution than that of A_1 with 5 stages (null hypothesis rejected, and alternative hypothesis supported, with *p-value* 3.0729e-145 < 0.01, U = 87422). Because A_{new} has a lower mode we can assert that the solutions it produces are significantly better than A_1, *for solutions with 5 stages*. However, for 6 stages, we found that no significant difference between the steps distributions (null hypothesis not rejected with p-value 0.4714 > 0.01, U = 164567).

Depending on the manufacturing DNA synthesis platform, it might be necessary to compromise either in the number of steps or the number of stages. Both algorithms, A_1 and A_{new} allows for the generation of a pareto-front of nondominated solutions from which one may choose plans with different characteristics. Figure 7 visualizes the potential pareto-front of stages and steps for iGEM-2008 with [1..4] extra slack. Crucially, for very large combinatorial libraries, only A_{new} would be able to produce such pareto front in a practical time.

Equivalent-Efficacy Efficiency Evaluation. If the greedy heuristics employed by A_{new} can exhibit equivalent efficacy—as measured by the biobjective function of equal *stages* **and** equal or fewer *steps*—compared to the much more computationally intensive dynamic programming approach of A_1 for a subset of real-world libraries, then it may be possible to relate common features of those particular libraries to easier instances of what we have shown to be hard problems and gain insight into which factors besides scale determine the difficulty of certain classes of *S-T* library assembly problems.

To evaluate the equivalent-efficacy efficiency of A_{new} relative to A_1, we measured the number of A_{new} runs, up to a cutoff of 1000, required to obtain at least one assembly plan that matched or surpassed the mode biobjective function value of 1000 A_1 plans. This measurement was repeated 10 times for each library in the experiment to obtain average values for a, the mean number of A_{new} runs necessary in order to obtain the equivalent efficacy of A_1. The speedup factor of A_{new} over A_1 is calculated as $b/(ca)$, where b and c are the mean running times of A_1 and A_{new}, respectively.

Table 2 shows the results of this evaluation on a new data set of real-world libraries from the CADMAD project, where all library targets are related variants of a similar construct and potential for DNA reuse is an intentional part of the design. Whereas, in the cases of the aggregate iGEM-2008 target set investigated above and any samplings of the iGEM submissions 2007−2012 data set used in the following section for the evaluation of algorithmic efficiency and scalability irrespective of desirable simultaneous manufacture, recurrent part combinations are more of a data set feature than part of an overall design.

Table 2. Bi-objective Function Values from A_1 (1000 runs) and A_{new} (When A_1 Mode Matched) for Real-World Libraries

library	statistic	A_1	A_{new}	A_{new} mean runs ±1 s.d. *a*	A_1 mean running time ±1 s.d. (s) *b*	A_{new} mean running time ±1 s.d. (s) *c*	speedup factor for equivalent efficacy *b*/(*ca*)
Phagemid ($k = 131$, $n_{max} = 14$)	best	(4, 204)	(4, 202)	$1.0 \pm 2.9 \times 10^{-4}$	2.965 ± 0.071	$0.007 \pm 4.148 \times 10^{-4}$	385
	mode	**(4, 207)**	**(4, 202)**				
	worst	(4, 236)	(4, 208)				
UKB ($k = 36$, $n_{max} = 21$)	best	(5, 126)	(5, 100)	$1.3 \pm 2.34 \times 10^{-4}$	5.984 ± 0.155	$0.0052 \pm 3.214 \times 10^{-4}$	885.2
	mode	**(5, 139)**	**(5, 100)**				
	worst	(5, 151)	(6, 114)				
UNOTT azurin ($k = 125$, $n_{max} = 7$)	best	(3, 170)	(3, 170)	$2.3 \pm 6.592 \times 10^{-3}$	2.725 ± 0.074	$0.0056 \pm 6.867 \times 10^{-4}$	211.57
	mode	**(3, 170)**	**(3, 170)**				
	worst	(3, 170)	(4, 170)				
UH1 ($k = 256$, $n_{max} = 6$)	best	(3, 300)	(3, 344)	**aborted after 1000 runs**	8.213 ± 0.517	$0.0093 \pm 5.155 \times 10^{-3}$	N/A (mode unmatched)
	mode	**(3, 304)**	**(3, 344)**				
	worst	(3, 304)	(3, 344)				
UH2 ($k = 96$, $n_{max} = 33$)	best	(6, 429)	(6, 274)	3.4 ± 0.030	325.423 ± 8.278	$0.023 \pm 1.554 \times 10^{-3}$	4161.42
	mode	**(6, 527)**	**(6, 285)**				
	worst	(6, 596)	(7, 354)				
ETHZ ($k = 625$, $n_{max} = 7$)	best	(3, 690)	(3, 690)	27.0 ± 0.484	51.549 ± 2.344	$0.027 \pm 3.026 \times 10^{-3}$	70.71
	mode	**(3, 690)**	**(3, 790)**				
	worst	(3, 690)	(4, 790)				

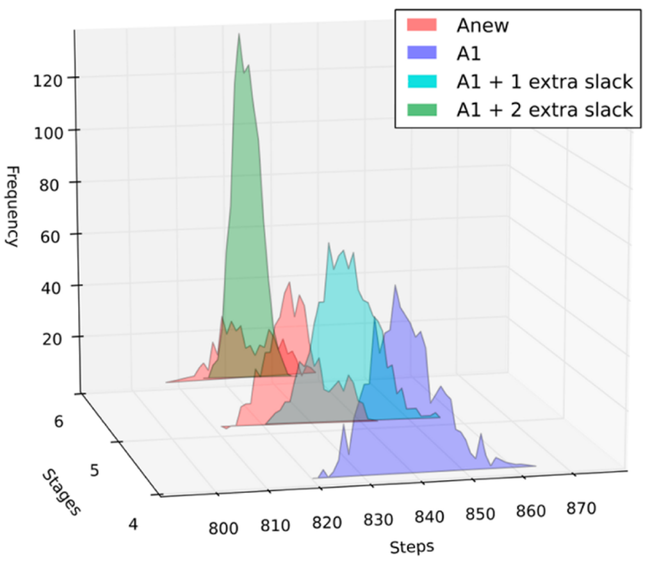


Figure 6. Distributions of steps for 1000 runs each of A_{new}, A_1, and A_1 with 1 and 2 extra slack on iGEM-2008.

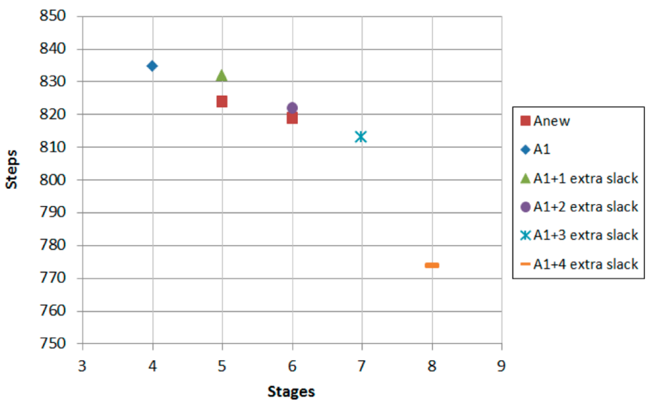


Figure 7. Pareto-front of (mode) stages vs steps for iGEM-2008 found by A_{new} and A_1 with extra slack from 0 to 4.

The biobjective function values for the Phagemid library remain as discussed in Table 1, except that we can now observe that A_{new} achieves an average 385-fold speedup over A_1 with at least equivalent efficacy. The best A_1 plan for the UKB library has (5, 126) vs (5, 100) at most on average for A_{new}, which achieves fewer steps every 1.3 runs on average and is 885-fold faster. The UNOTT azurin library again shows that A_{new} can yield deeper than necessary assembly graphs on occasion, such that A_{new} can find the known optimal (3, 170) solution once every 2.3 runs on average, in 1/200th of the time it takes A_1.

For the UH1 library, however, A_{new} cannot match A_1 in at least 1000 runs. In fact, probably never, since the best, mode and worst for A_{new} are all equivalent, and likely the same or very similar collections of concatenation steps. Although superficially similar to the ETHZ library design, the UH1 Cartesian product is without shared linkers, which presents a problem for A_{new}, as all pairs occur with equal frequency in the first stage, making greedy concatenation of subsets of pairs across targets in this scenario an ineffective strategy compared to the per-target concatenation approach of A_1. These poor results for A_{new} on UH1 are in sharp contrast to those for UH2, where A_{new} significantly outperforms A_1 in terms of both efficacy and efficiency, reducing the number of steps by almost a factor of 2 on average and requiring 1/4160th of the running time. The difference in computational efficiency can be accounted for largely by the greater n_{max} of UH2, as the runtime complexity of A_1 has been shown analytically to grow more rapidly with the number of parts per target, than the number of targets. The particularly high speedup factor is due to A_{new} on average finding a better result than the mode of A_1 within the time required to execute just four runs, with a much shorter mean running time.

For the ETHZ library, the best, mode, and worst for A_1 are all equal (and possibly optimal), so to match the average result is also to match the best, which A_{new} does on average, every 27 runs. As it takes just 0.027 s compared to 51.549 s for one run of A_1, a 70-fold speedup is achieved. Overall, we can conclude that A_{new} often matches and sometimes surpasses the quality of assembly plans produced by A_1, for several quite-distinct

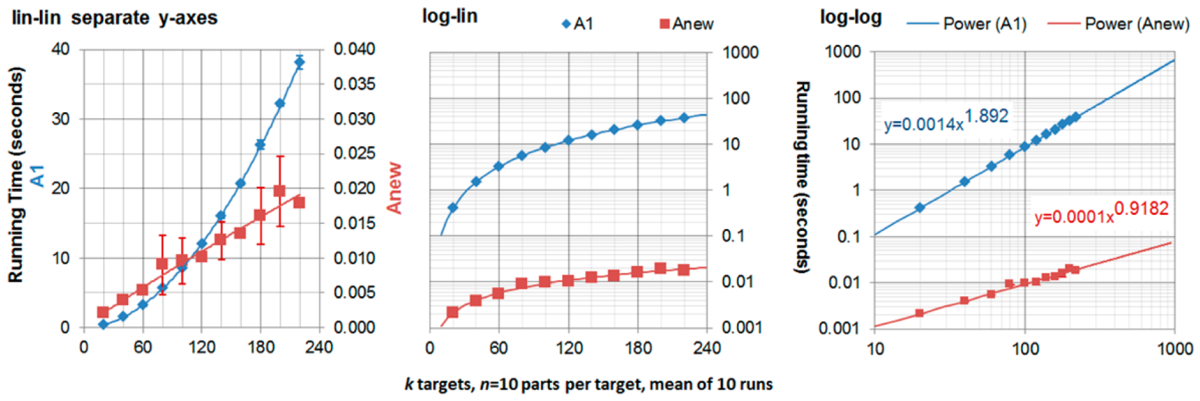


Figure 8. Running time vs k, the number of targets. A_1 is shown as blue diamonds, and A_{new} as red squares. In the left lin−lin plot A_1 grows faster than A_{new} which is linear despite some large variances. Separate y-axes facilitate the visualization of the trend for A_{new} that would otherwise appear flat due to differences in scale: A_1 is plotted on the left y-axis and A_{new} on the right, the values of which are 3 orders of magnitude smaller than for A_1. The central log−lin plot, allows the timings for both algorithms to be plotted on the same axes, confirming that for $k = 120$ and $n_{max} = 10$ (just after the lines intersect in the lin−lin plot) A_{new} is at least 1000 times faster. The right log−log plot estimates the exponent of k to be ∼2 for A_1 and almost ∼1 for A_{new}.

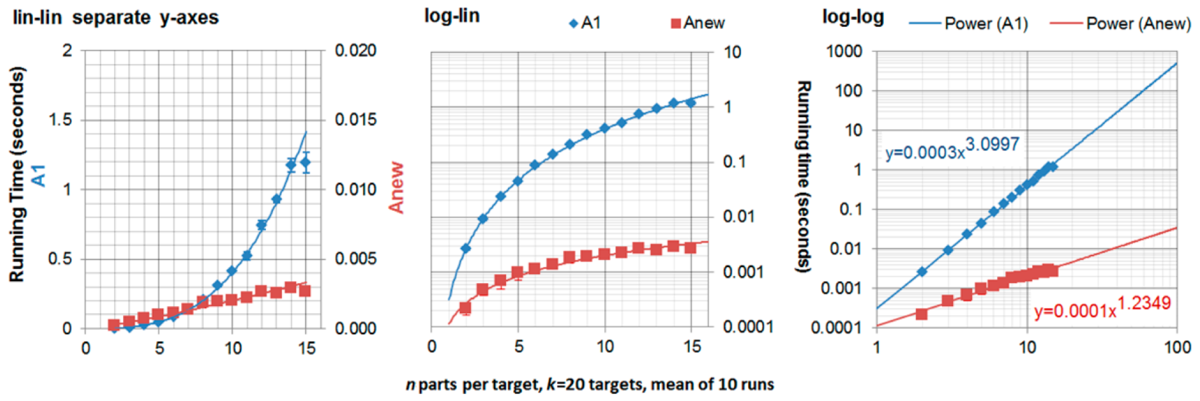


Figure 9. Running time vs n, the number of parts per target. The presentation is essentially identical to that of Figure 8, except that in the lin−lin plot the scale of the right y-axis (A_{new}) is 2, not 3, orders of magnitude smaller than the left (A_1). The interpretation differs only in that the right log−log plot estimates the exponent of n_{max} to be just over 3 for A_1 and 1.235 for A_{new}. Since all of the targets for a given data point have the same value of n, this represents the worst case for algorithms with runtime complexity based on n_{max}.

real-world libraries. Additionally, the time required by A_{new} to generate an assembly plan of equivalent quality to A_1 can be in excess of 3 orders of magnitude smaller, and an at least an order of magnitude speedup is achieved for all but one of the data sets here.

In the next section, we confirm this large improvement in efficiency, using an artificial data set to investigate the independent effects of library size (k) and maximum target length (n_{max}) on running time growth for both algorithms.

Efficiency Evaluation. The theoretically determined runtime complexity of $O(k^2 n_{max}^3)$ for A_1 is the product of exponents of the library variables k (number of targets) and n_{max} (number of parts in largest target), making it likely that the runtime complexity of A_{new} can also be expressed in terms of these variables. Therefore, to gather empirical evidence of the runtime complexity of A_{new}, we measured running times against k and n_{max} independently, sampling appropriate 'libraries' from the iGEM_submissions_2007−2012 target set. For the best estimation of worst-case complexity, all targets should have n equal to n_{max}. Estimates of the exponent for each variable, and for both algorithms, were taken as the gradient of the power trend lines on the respective log−log plots.

To measure running time response to k, an n of 10 was chosen, yielding a population of 224 potential targets. For each value of k from 20 to 220 in increments of 20, ten k-sized samples were run once with both algorithms and the mean running times calculated. The results are plotted in Figure 8. (Standard deviations are plotted for every data point in both Figures 8 and 9 but are visible only on the plots where both axes are linear and not in those with logarithmic running time y-axes.)

To measure running time response to n_{max}, we established a range for n from 2 to 15 for which a minimum of 35 targets were available. Since these targets were grouped by n and not necessarily related or intended to be assembled together, rather than selecting the maximum 35 targets for each n and taking the mean running time of multiple runs on only those targets, we decided instead to effectively generate ten different libraries for each n by taking 10 $k = 20$ samples for each value of n, performing a single run on each sample and taking the average running time (i.e., each algorithm ran on all 'libraries'). The results are plotted in Figure 9.

Using Figures 8 and 9, we estimate an empirically determined runtime complexity for A_{new} of $O(k^{0.92} n_{max}^{1.235})$ and for A_1 of $O(k^{1.9} n_{max}^{3.01})$, confirming its theoretical time complexity. In the Methods section, we prove that the time complexity of A_{new} is in fact $O(kn_{max})$, or linear with respect to both k and n_{max}, suggesting that a data set with values of n_{max} somewhat

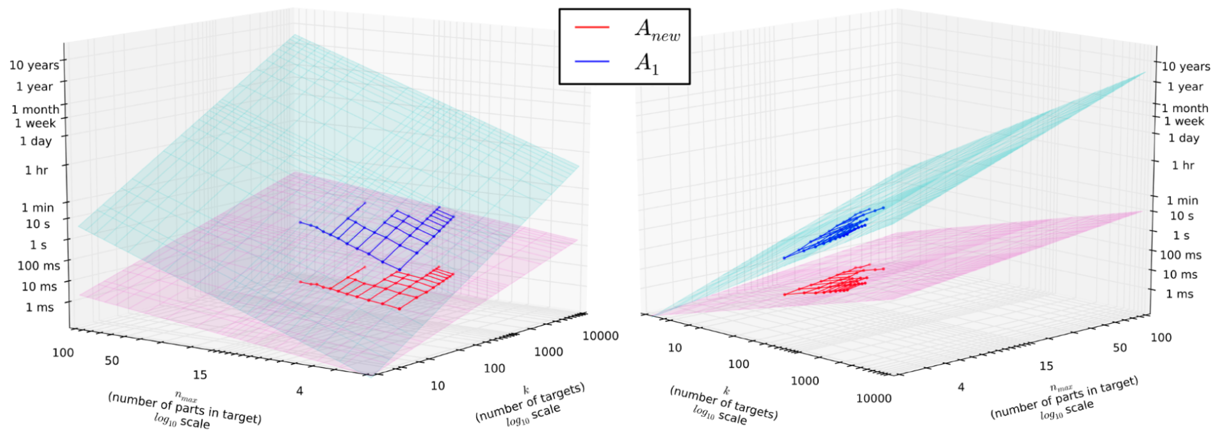


Figure 10. 3-Dimensional log−log−log plot showing the response of running time to k (targets) and n (parts per target). Each point is the mean of 10 runs (standard deviations not shown). The lines joining the points are aligned to planes, cyan for A_1 and magenta for A_{new}, that extrapolate the theoretical runtime complexities to estimate the running times for larger values of k and n_{max}.

larger than 15 would yield log−log plots of empirical timings with gradients closer to 1.

Figure 10 plots mean running times for the 4242 target iGEM_submissions_2007−2012 data set, grouped by n, on planes extrapolated from the asymptotic time complexities of the algorithms. We can observe from that the growth of running time responses to k (Figure 8) and n_{max} (Figure 9) for algorithms A_{new} and A_1 are replicated across both ranges of values. The height separation and difference in the two-dimensional gradient of the planes definitively demonstrates that A_{new} is always faster than A_1, with much improved scalability: at the maximal values $k = 10000$ and $n_{max} = 100$, a single run would require 10 s for A_{new} versus several years for A_1.

Discussion. Two computational problems arise from the need to identify reusable parts within a given set of desired DNA sequences: identifying common building blocks and devising a plan requiring the fewest concatenations to construct the targets. The latter is addressed in this paper, while the former is equivalent to rediscovering the building blocks (such as words) of natural language texts when the original words have all been joined together and has been identified before as the Minimum String Cover (MSC) problem.[17] In the context of DNA libraries these words must be sufficiently long to facilitate manipulation by PCR. Attempts at this problem show some promising results but so far cannot effectively scale up to lengths from a few hundred to several thousand base pairs required for most DNA libraries.[18] Combinatorial DNA library designs, particularly in the field of synthetic biology, are commonly derived from registries of accessible primitive (and composite) parts that already constitute a quite reasonable MSC solution.

Combinatorial DNA library manufacturing technologies that maximize DNA reuse can increase throughput while simultaneously reducing error rates and costs for both producers and researchers. We have presented a new algorithm predicated on the maximization of DNA reuse that devises DNA library assembly plans preserving quality comparable to the state-of-the-art (requiring fewer assembly steps/intermediates but occasionally more stages) with a vast improvement in running time and scalability.

A limitation of our algorithm, and also of A_1, is that only binary concatenation steps are considered. It is directly suited to assembly methods capable of joining arbitrary DNA parts or composite parts whose subparts have compatible ends. n-ary assembly methods such as Gibson assembly,[19] that concatenate two or more parts in a single step, should be incorporated, as these have the capacity to further reduce stages. Conversely, restricting our algorithm to binary concatenation enabled a meaningful comparison of solution quality with A_1, the only other previously published method.

The presented simple pairs ranking by count can be expanded to more complex pair scoring functions. Taking into account various production criteria such as biochemical ease of assembly (fragment length, low complexity regions, etc.), regional robustness in downstream processing or presence in longer repeated elements (trios, quads etc.) calls for a multifragment assembly. Other considerations such as mounting input fragments on primers in the final assembly process or grouping multiple binary assemblies into a single n-ary assembly can be added as a pre- or post-process if desirable.

Theoretically, n-ary assembly methods, allow library manufacturers to eschew the assembly order problem altogether, producing all k targets in one stage and k steps; without reuse and assuming no mixed products. Today's methods, however, suffer from decreasing confidence as the fragments multiply and can cope with the assembly of only limited number of fragments at once.

Empirical running time analysis of both algorithms showed that A_{new} outperforms A_1 by at least several orders of magnitude on real-world libraries and that running times grow at a slower rate as either the number or length of targets library parts increases. Experiments with fixed numbers of library targets and their sizes show that the running time of A_{new} scales linearly with both the number of target parts k and the maximum number of component parts n_{max}, confirming the $O(kn_{max})$ runtime complexity theoretically determined in the analysis of the recursive formulation in Listing 2. With the same fixed-n_{max} libraries, we empirically confirmed the $O(k^2 n_{max}{}^3)$ theoretical runtime complexity of A_1 to be representative of its actual performance.

In practice, DNA library manufacturers want to be able to select from a variety of viable assembly plans, optimizing various criteria: fewest operations/generations, lower consumable costs, avoiding problematic intermediates, etc. As such, it can be more intuitive to think of speedup factors in terms of number of library plans different assembly algorithms can generate in an allotted time frame. The benefit of the speedup we report is that the greedy heuristic can generate, in the case

```
T′ ← set of targets designed with available parts in S                                      1
stages ← empty list to populate with planned steps in run (each stage is a set of steps)    3
function plan_assembly_stages_recursive(T′, stages)                                         5
    1. count overlapping pairs in T′ and identify trios in single pass; O(N)                7
    2. map pairs to containing trios (and vice versa); O(N)                                16
    3. shuffle O(N) then stably sort O(s(N)) pairs to reorder within rankings              24
    4. add non-excluded pairs to stage and exclude overlapping using maps from 2; O(N)     29
          if 0 steps in stage then return stages else add stage to stages O(1)             37
    5. simulate assembly stage updating T′ sequences in-place; O(N)                        39
       plan_assembly_stages_recursive(T′, stages) 6. recursively populate stages            -
```

Listing 2. A_{new}, high-level, recursive pseudo-code; worst case runtime complexity.

of the UKB library for example, ~800 times the number of assembly plans as the previous state-of-the-art algorithm for the same time cost.

Our results show that in terms of the biobjective (*stages*, *steps*) function A_{new} was either as good as or better than A_1 for most of the DNA library designs used to test the algorithms. A_{new} is still complemented by A_1 though, as we also found that depending on the library A_{new} cannot always produce solutions with minimal stages, and that in the case of one library (UH1) that suboptimal plans were obtained when the composition of targets is such that there is not sufficient gradient in terms of more frequently occurring pairs for the heuristics to have a positive benefit on planning efficacy.

The impact of the steps versus stages trade-off depends mainly on the DNA manufacturing platform. Where additional stages can be quite costly in terms of time and risk of error, biochemical concatenation steps reflect the majority of reagent costs. In many cases, the throughput required for many steps can only be achieved by queuing steps on multiple platform cycles, further increasing time and operation costs. In the future, closed manufacturing systems, where the output of one stage can be used directly as the input of the next without a large time lag, may prioritise reducing the number of steps to focus more in line with maximizing DNA reuse.

We showed that the abstract computational problem *bounded-depth min-cost string production* (BDMSP) underlying efficient combinatorial DNA library assembly is NP-hard and APX-hard. Since our new algorithm does not limit the number of stages, but rather prejudices fewer stages by applying steps globally, obtaining solutions with a variety of stages dependent on input, it addresses the broader, unbounded problem (MSP), which *may* also be NP-hard (an investigation we will pursue in our future work). In its current form, our method does not guarantee optimal solutions to the very difficult problem of devising assembly plans. Nevertheless, through a combination of greedy heuristics and randomization, our new algorithm consistently produces a tight distribution of high quality plans, and can therefore be run repeatedly to quickly generate a set of solutions that can be filtered or augmented according to additional criteria relevant to automated manufacturing of these large combinatorial DNA libraries. From both design and production perspectives, it can give quick and reliable feedback, often in real-time, as to number of necessary assembly steps and their identities, a proxy for the effort and costs involved.

■ METHODS

Running Time Measurements. The running times reported here were obtained with Python 2.7.3 64-bit under Linux 3.2.0-4-amd64 with 32GB RAM using single cores of a i7-2670 2.2 MHz processor.

Runtime Complexity Analysis Using Recurrences. Listing 2 outlines the runtime complexity for a recursive implementation of A_{new}, using the following definitions: Let T be the set of target parts, S the set of available primitive parts, and T' the targets in T composed of parts in S.

Define pair as two consecutive primitive parts in T'.

$N = kn_{max}$, where k is the number of target parts and n_{max} is the maximum number of component parts in any target.

$s(N) = sorting(N)$, where depending on the choice of sorting algorithm,[20] best, average and worst case parameters can vary. Our implementation can use either Python's built-in Timsort (the default), with best $\Theta(n)$ vs $\Theta(n \log n)$ average and worst case performance, or a simple tailored counting sort with worst case $O(n)$ performance.

In terms of running time, the Master theorem[20] concerns recursive algorithms that can be represented in the form of $T(N) = aT(n/b) + f(n)$ where a is the number of subproblems in the recursion, n is the size of the problem and n/b is the size of each subproblem. According to case 3 of the theorem, if $f(n) = \Theta(n^c)$ when $c > \log_b a$, then $T(n) = \Theta(f(n))$. For A_{new}, $a = 1$ and $1 < b \leq 2$ (the best case being 2 when half of all pairs are steps) hence A_{new} always falls under case 3, as $C > \log_{[1..2]} \mathbf{1}$, therefore $T(N) = O(s(N))$. In other words, the worst case complexity of A_{new} is dependent on the worst case complexity of the sorting algorithm used. Profiling showed the sorting step to not be a bottleneck: 0.1% of running time consistently for various libraries. Empirically, substituting Timsort with a worst-case linear time counting sort led to marginally longer running times. Theoretically speaking, as the space required for the mapping data structures grows linearly $\Theta(n)$ with the size of input, space-time considerations for the sorting algorithm of choice may shift for linear-time/linear-space algorithms[20,21] without sacrificing space efficiency. In summary, as the runtime was found to be bound only by the sorting function, and as linear-time sorting functions can be implemented without increasing space requirements, it is safe to say the algorithm has linear run time, that is, $O(s(N)) = O(N)$.

Data Set Curation. Benchmark libraries small artificial, Phagemid, and iGEM-2008 were reused from ref 13, for acceptance testing of A_1 as a proxy for A_{mgp}, and efficacy comparison with A_{new}.

For equivalent-efficacy efficiency comparison, the following libraries from the CADMAD project (FP7 STREP 265505) were used:

- UKB has 36 targets, in several subsets, where a restriction site flanked element is either present or missing;
- UNOTT azurin is a set of 125 targets, where 3 partial coding sequences that putatively form stem-loop structures with an AGGA *Rsma*-binding motif are replaced by not AGGA-containing alternative back-translations of the same coding sequence, interspersed with constant coding regions;
- UH1 targets are the full factorial $4 \times 4 \times 4 \times 4$ Cartesian product of 4 protein domains from 4 eukaryotic orthologs;
- UH2 follows a similar design of experiments methodology, but samples only 96 targets from a space of 2^{12}

Table 3. Libraries and Target Set Used to Evaluate Assembly Algorithms

library/target set	k |targets|	n_{max} |most parts|	minimum stages[a]	experiments used in	related figures/tables
small artificial[b]	5	5	3	A_1 vs A_{mgp} validation	Table 1
				A_1 vs A_{new} efficacy	
Phagemid[b]	131	10	4	A_1 vs A_{mgp} validation	Tables 1 and 2
				A_1 vs A_{new} efficacy	
				A_1 vs A_{new} equivalent-efficacy efficiency	
iGEM-2008[b]	395	14	4	A_1 vs A_{mgp} validation	Table 1; Figures 5, 6, and 7
				A_1 vs A_{new} efficacy	
UKB[c]	36	21	5	Assembly graph visualizations	Figures 1 and 4; Table 2
				A_1 vs A_{new} equivalent-efficacy efficiency	
UNOTT azurin[c]	125	7	3	A_1 vs A_{new} equivalent-efficacy efficiency	Table 2
UH1[c]	256	6	3	A_1 vs A_{new} equivalent-efficacy efficiency	Table 2
UH2[c]	96	33	6	A_1 vs A_{new} equivalent-efficacy efficiency	Table 2
ETHZ[c]	625	7	3	A_1 vs A_{new} equivalent-efficacy efficiency	Table 2
iGEM submissions 2007–2012[d]	4242	31	5	A_1 vs A_{new} efficiency	Figures 8, 9, and 10; Table 4

[a]Calculated as $ceil(\log_2(n_{max}))$. [b]Libraries from.[13] iGEM-2008 is somewhat arbitrary in that the targets are not all related variants. [c]CADMAD libraries for Universität Bonn (UKB), University of Nottingham (UNOTT), University of Helsinki (UH), and ETH Zurich (ETHZ). [d]A new data set curated from BioBrick parts submitted to the Registry of Standard Biological Parts after the annual iGEM competition, containing unique composite parts from years 2007 to 2012 (containing 505, 885, 1133, 681, 597, 552 parts, respectively).

Table 4. Breakdown of k Values by n for All Years of iGEM_submissions_2007–2012 Combined

n[a]	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k	788	746	462	843	296	272	129	122	224[b]	65	50	67	35	83
n (continued)[c]	16	17	18	19	20	21	22	23	24	25	26	28	29	31
k (continued)[c]	9	7	7	10	5	6	1	3	2	4	2	1	2	1

[a]$k \geq 20$ and $2 \leq n_{max} \leq 15$; Figure 9. [b]Figure 8. [c]Figure 10.

(4096) sequence, the product of 12 codon changes with 2 potential codons in each position;

- ETHZ is a $5 \times 5 \times 1 \times 5 \times 1 \times 5 \times 1$ set of 625 targets composed of 3 genes interspersed with combinations of promoters and ribosome binding sites.

A further data set of annual iGEM-submissions 2007–2012 provided sufficient number of composite BioBrick parts to investigate independently the effect of k and n_{max} on running times, that is, a range of n values with reasonable k, and hence a choice of k with equal n. Table 3 shows the libraries and target set used to evaluate assembly algorithms.

The iGEM_submissions_2007–2012 data set was scraped from the Registry of Standard Biological Parts' Libraries page (http://parts.igem.org/assembly/libraries.cgi) by following the iGEM-submissions-by-year links. We obtained the submitted part ID and list of subparts for each composite part. Duplicate composite parts (with differing IDs) were removed so that each part was counted only once. The complete set of 4242 unique composite targets was binned by target size n to identify for which values of n was k sufficiently large to separately measure running time response to ranges k and n_{max} values (and ensure for all targets $n = n_{max}$). These values and the corresponding figures are collected in Table 4.

APPENDIX I: COMPUTATIONAL COMPLEXITY ANALYSIS FOR THE BOUNDED-DEPTH MIN-COST STRING PRODUCTION (BDMSP) PROBLEM

In this appendix we analyze the computational complexity of the Bounded-Depth Min-Cost String Production (BDMSP) Problem. We first give a high level introduction to the techniques used for analyzing computational complexity, we then provide the core theorem and finally we discuss its implications.

Essential Computational Complexity

We briefly review some basic and well-known aspects of computational complexity. For more rigorous treatment, see for example Garey and Johnson's seminal book.[15] Our review will use BDMSP as a running example. BDMSP is an optimization problem. However, it can equivalently be viewed as a decision problem. The input to this computational problem is a set of source strings, a set of target strings, an upper bound on the allowable depth, and an upper bound on the allowed number of steps. A solution to the problem is a production plan (a.k.a. assembly plan), and a solution is feasible if indeed it generates all target strings starting only from source strings, and it satisfies the step and depth requirements. Given a proposed solution, checking its feasibility is a computationally easy task (it can be done in time polynomial in the input size), which places BDMSP in the computational class of NP (nondeterministic polynomial time). What we would like to determine is whether BDMSP is in the complexity class P, namely, given an instance of the problem, can a solution always be found efficiently (in polynomial time), if one exists. This requires that we either design an algorithm for BDMSP and prove that it runs in polynomial time, or prove that no such algorithm can exist. Current computational complexity techniques are not able to establish either of these two options, but they can strongly indicate that BDMSP is not in P. This is done by showing that BDMSP is NP-hard (informally meaning that its structure is sufficiently rich so as to encode in it every possible NP problem), and thus if BDMSP were in P this would establish that P = NP (which is widely believed not to be true, and remains the most famous open question in complexity theory for many decades). To establish that BDMSP is NP-hard it

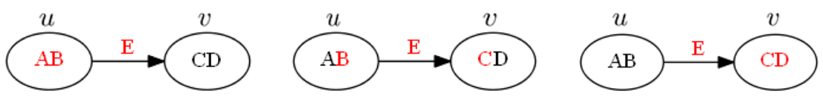


Figure 11. Every edge describes three composite parts of length 3 {*ABE, BEC, ECD*} shown in red.

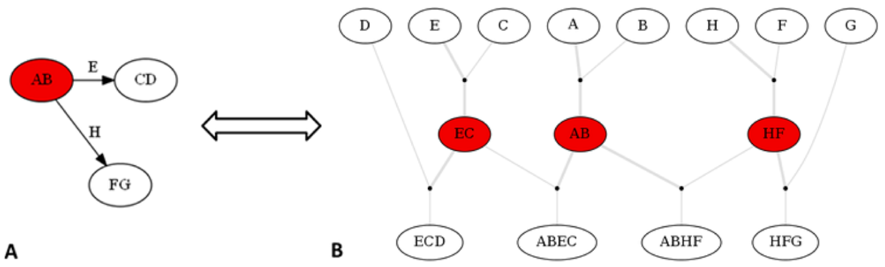


Figure 12. Bidirectional VC to BDMSP reductions: (A) graph of targets {*ABE, BEC, ECD, ABH, BHF, HFG*} emphasizing the minimal vertex cover node, red; (B) optimal assembly graph for targets emphasizing the intermediate nodes that result from the minimal vertex cover, red nodes.

suffices to show that it encodes just one other NP-hard problem (in technical terms, one other NP-hard problem reduces to BDMSP), and then by transitivity all NP-hard problems reduce to BDMSP. In our proof, we reduce the well known NP-hard problem minimum vertex cover to BDMSP.

Theorem: *BDMSP is NP-hard and APX-hard. Furthermore, this holds even when S is composed only of single characters and d = 2, and there is a total order among characters such that in every string of T this order is respected.*

Proof: By a reduction from vertex cover (VC). Let G be an arbitrary graph of maximum degree 3, with n vertices and $m \leq 3n/2$ edges, in which we need to determine if there is a vertex cover of size k. Number the vertices of G from 1 to n and direct every edge of G from its lower numbered end point to its higher numbered end point (thus there are no directed cycles). Consider a finite alphabet of size $m + 2n$. The set of these characters will serve as S for BDMSP. Label each vertex by two distinct characters and each edge by one distinct character. Every directed edge (u,v) gives rise to three strings in T as follows. Suppose that u is labelled by AB, that v is labelled by CD, and that the edge (u,v) is labelled by E. Then, the corresponding three target strings are ABE, BEC, and ECD, shown in Figure 11. The total number of target strings is $3m$. We limit the depth d of the production process to be 2.

As every target string contains three characters, at least two of them need to be concatenated already in the first stage, as otherwise the depth will exceed 2. Hence, among the three strings ABE, BEC, and ECD discussed above, we must perform at least two concatenations in the first stage: one a *vertex concatenation* such as AB, and the other an *edge concatenation* such as EC. The edge concatenation (EC) must involve E and hence does not serve target strings of any edge other than (u,v). The vertex concatenation (AB) may serve other target strings as well.

In the second stage, at least two concatenation operations are needed per edge (concatenating AB with EC gives $ABEC$ that contains both ABE and BEC, and concatenating EC with D gives ECD).

VC of Size k Implies BDMSP of cost 3m + k. Given a vertex cover VC of size k, perform k vertex concatenations in the first stage, each corresponding to one vertex from the vertex cover VC. In addition, for every directed edge perform one edge concatenation. In the example above, if u is in the vertex cover generate AB and EC, otherwise generate CD and BE. The total number of steps is $m + k$ (in the first stage) plus $2m$ (in the second stage), for a total of $3m + k$.

BDMSP of Cost 3m + k Implies VC of Size k. Given a solution to the BDMSP instance of depth two with $3m + k$ steps, every character that labels an edge appears in at least one intermediate string (because it is a middle character in some target string). Without loss of generality, we may assume that it is not contained in two intermediate strings. (In the example above, if we have the intermediate strings BE and EC, they can be replaced by BE and CD without increasing the cost of the solution.) Hence, at least one of the end points of the edge must contribute an intermediate string as well. This means that the set of intermediate strings that do not contain edge labels corresponds to a vertex cover. As stage two requires at least $2m$ steps and stage one has at least m edge concatenations, the vertex cover is of size at most $3m + k - 3m = k$.

Both NP-hardness[14] and APX-hardness[15] of BDMSP follows from that of vertex cover in bounded degree graphs.

Figure 12A shows an example of how the solution to the equivalent VC problem solves MSP, and *vice versa*, for the library $T = \{ABE, BEC, ECD, ABH, BHF, HFG\}$ over the set of primitive parts $S = \{A, B, C, D, E, F, G, H\}$. The solution to VC for the given graph is AB, resulting in choosing the intermediate part AB for the node in the vertex cover and EC HF for the edges.

We remark that if depth is not limited, that is, the problem is unbounded, then the target set in instances of the proof can be generated in $n + 2m < 3m$ steps. In stage one, do all n vertex concatenations. In stage two, do exactly one concatenation per directed edge (e.g., produce ABE but not ECD), producing one of the target strings associated with the edge. In stage three, using the previously produced target string as an intermediate string, do the other concatenation for every directed edge, creating the two remaining target strings in parallel. For example, concatenate ABE and CD, producing $ABECD$ that contains both BEC and ECD (Proving the NP-hardness of MSP is not trivial and exceeds the scope of our efforts here, which is concerned with practical algorithms for very large combinatorial DNA assembly plans).

Implication

The consequences of NP-hardness are that there is no polynomial time algorithm that solves BDMSP optimally on every input instance (unless P = NP). This does not imply that there are no efficient algorithms that solve some of the instances of BDMSP correctly, and perhaps even all those instances that one would encounter in practice. Computational complexity theory currently does not provide the tools for addressing such questions. Another question one may ask is whether there is an efficient algorithm that solves BDMSP almost optimally (rather than exactly optimally) on all instances. These types of questions are addressed in computational complexity theory through the notion of APX-hardness (APX stands for approximation). Minimum vertex cover is known to be APX-hard, meaning that there is a limit to how well it can be approximated by polynomial time algorithms, unless P = NP. Our reduction from minimum vertex cover to BDMSP is an approximation preserving reduction, implying that BDMSP is also APX-hard. This means that there is some constant r such that unless P = NP, there is no polynomial time algorithm that finds solutions for BDMSP that are approximately optimal, not even if one allows a multiplicative slackness factor of r in the number of production steps.

■ ASSOCIATED CONTENT

S Supporting Information

Publicly accessible web application that rapidly computes near optimal plans for combinatorial DNA library assembly is available at http://www.dnald.org/planner/. We also published implementations of both A_{new} and A_1, the parts library data sets and empirical performance comparison scripts. This material is available free of charge via the Internet at http://pubs.acs.org.

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: natalio.krasnogor@newcastle.ac.uk.
*E-mail: ehud.shapiro@weizmann.ac.il.

Author Contributions

[†]J.B. and O.R. are joint first authors.

Author Contributions

J. Blakes and O. Raz contributed the greedy heuristic, algorithm implementations, complexity analysis, designed experiments, performed experiments, and wrote the manuscript. U. Feige contributed the *bounded-depth min-cost string production* (BDMSP) formulation and computational hardness results, and wrote the manuscript. J. Bacardit advised on pareto-front efficacy analysis and statistical tests. P. Widera built the web application associated with the main algorithm and helped to analyze the experimental data. T. Ben-Yehezkel contributed the problem description and DNA assembly advice. E. Shapiro and N. Krasnogor secured the funding, conceived the study, contributed to experimental and complexity analysis, designed experiments, and wrote the manuscript.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

This work was supported by the European Union FET project CADMAD (FP7 STREP 265505). N.K. also received support from the Engineering and Physical Sciences Research Council (EPSRC) grants ROADBLOCK (EP/I031642/1), AUDACIOUS (EP/J004111/1), and TAURUS (EP/L001489/1).

■ REFERENCES

(1) Wetterstrand K. A. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). https://www.genome.gov/sequencingcosts/ (accessed 2014/03/25).

(2) Stratton, M. R., Campbell, P. J., and Futreal, P. A. (2009) The cancer genome. *Nature 458*, 719−724.

(3) Carr, P. A., and Church, G. M. (2009) Genome engineering. *Nat. Biotechnol. 27*, 1151−1162.

(4) Ellis, T., Adie, T., and Baldwin, G. S. (2011) DNA assembly for synthetic biology: From parts to pathways and beyond. *Integr. Biol. (Camb.) 3*, 109−118.

(5) Czar, M. J., Anderson, J. C., Bader, J. S., and Peccoud, J. (2009) Gene synthesis demystified. *Trends Biotechnol. 27*, 63−72.

(6) Tian, J., Ma, K., and Saaem, I. (2009) Advancing high-throughput gene synthesis technology. *Mol. Biosyst. 5*, 714−722.

(7) Ryan, D., and Papamichail, D. (2013) Rational design of orthogonal libraries of protein coding genes. *ACS Synth. Biol. 2*, 237−244.

(8) Raab, D., Graf, M., Notka, F., Schodl, T., and Wagner, R. (2010) The GeneOptimizer algorithm: Using a sliding window approach to cope with the vast sequence space in multiparameter DNA sequence optimization. *Syst. Synth. Biol. 4*, 215−225.

(9) Girdea, M., Noe, L., and Kucherov, G. (2010) Back-translation for discovering distant protein homologies in the presence of frameshift mutations. *Algorithms Mol. Biol. 5*, 6.

(10) Tang, L., Gao, H., Zhu, X., Wang, X., Zhou, M., and Jiang, R. (2012) Construction of ″small-intelligent″ focused mutagenesis libraries using well-designed combinatorial degenerate primers. *Biotechniques 52*, 149−158.

(11) Shabi, U., Kaplan, S., Linshiz, G., Ben-Yehezkel, T., Buaron, H., Mazor, Y., and Shapiro, E. (2010) Processing DNA molecules as text. *Syst. Synth. Biol. 4*, 227−236.

(12) Linshiz, G., Stawski, N., Poust, S., Bi, C., Keasling, J. D., and Hillson, N. J. (2012) PaR-PaR Laboratory Automation Platform. *ACS Synth. Biol. 2*, 216−222.

(13) Densmore, D., Hsiau, T. H. C., Kittleson, J. T., DeLoache, W., Batten, C., and Anderson, J. C. (2010) Algorithms for automated DNA assembly. *Nucleic Acids Res. 38*, 2607−2616.

(14) Karp, R. M. (1972) *Reducibility among Combinatorial Problems*, Springer, New York.

(15) Gary, M. R., and Johnson, D. S. (1979) Computers and Intractability: A Guide to the Theory of NP-completeness, WH Freeman and Company, New York.

(16) Durstenfeld, R. (1964) Algorithm 235: Random permutation. *Commun. ACM 7*, 420.

(17) Hermelin, D., Rawitz, D., Rizzi, R., and Vialette, S. (2008) The Minimum Substring Cover problem. *Inf. Comput. 206*, 1303−1312.

(18) Canzar, S., Marschall, T., Rahmann, S., and Schwiegelshohn, C. (2011) Solving the minimum string cover problem. *2012 Proc. 14th Workshop Algorithm Eng. Exp. (ALENEX)*, 75−83.

(19) Gibson, D. G., Young, L., Chuang, R. Y., Venter, J. C., Hutchison, C. A., 3rd, and Smith, H. O. (2009) Enzymatic assembly of DNA molecules up to several hundred kilobases. *Nat. Methods 6*, 343−345.

(20) Leiserson, C. E., Rivest, R. L., Stein, C., and Cormen, T. H. (2001) *Introduction to Algorithms*, 2nd ed., pp168−170, The MIT Press, Cambridge, MA.

(21) Beniwal, S., and Grover, D. (2013) Comparison of various sorting algorithms: A review. *Int. J. Emerging Res. Manage. Technol. 2*, 83−86.