A New Algorithm for Combinatorial DNA Library Assembly

Jonathan Blakes
School of Computer Science
University of Nottingham
Nottingham NG8 1BB, UK
+44 115 84 68403
jonathan.blakes@nottingham.ac.uk

Ofir Raz
Department of Computer Science and Applied Math.
Weizmann Institute of Science
Rehovot 76100, Israel
+972 8 934 4506
ofir.raz@weizmann.ac.il

Natalio Krasnogor *
School of Computer Science
University of Nottingham
Nottingham NG8 1BB, UK
+44 115 84 67592
natalio.krasnogor@nottingham.ac.uk

Ehud Shapiro *
Department of Computer Science and Applied Math.
Weizmann Institute of Science
Rehovot 76100, Israel
+972 8 934 4506
ehud.shapiro@weizmann.ac.il

ABSTRACT

We describe a novel greedy algorithm for computing near-optimal DNA assembly graphs and show empirically that it runs in linear time, enabling almost instantaneous planning of DNA library sizes exceeding the capacity of today's biochemical assembly methods. We compare assembly graph quality and algorithmic performance to the results obtained in [1], demonstrating that they are significantly faster to obtain and equivalent to the best results for DNA library assembly with intermediate reuse found in the literature.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *sequencing and scheduling, computations on discrete structures*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

DNA assembly, combinatorial DNA libraries, gene design, genomic libraries, synthetic biology.

1. INTRODUCTION

Combinatorial DNA libraries, composed of natural and synthetic biological parts, can be assembled in an efficient manner. This is achieved by recognising that certain combinations of primitive and composite parts can occur multiple times within a library. Thus ordering the necessary assembly steps into stages so that these common, reusable intermediates are assembled first, in parallel, allows for reuse of correctly assembled composites between library members.

* Corresponding authors.

Fifth International Workshop on Bio-Design Automation (IWBDA 2013), July 12–13, 2013, London, United Kingdom.

1.1 Formal Problem Definition and Example

Let *S* be a set of available DNA parts and *T* a set whose elements are ordered sequences of elements in *S* which constitute a DNA library. Define a library assembly graph (as in Figure 1) as a hierarchical clustering, representing the binary convergence of nodes in *S* to nodes in *T*.

Given an assembly graph quality/cost function(stages,steps) that describes the ratio in penalties between the number of stages (depth of the construction graph) and number of intermediate steps (nodes that are neither in *S* nor in *T*), the problem is finding the assembly graph with optimal quality/minimal cost. In what follows we use the same cost function as [1] which seeks to minimize the number of stages and then the number of steps.

For *S*={A,B,C,D,E} and *T*={ABE,ABDE,ACDE,ADE} Figure 2 shows a suboptimal (2,8) and the optimal (2,7) assemblies.

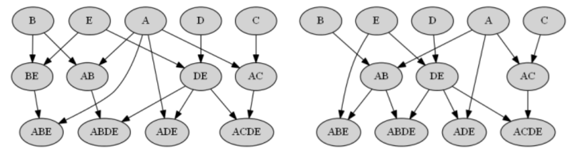

Figure 1: Suboptimal (left) and optimal library assembly.

1.2 State-of-the-Art

The authors of [1] contributed an iterative refinement algorithm based on dynamic programming for finding the optimal binary assembly tree (minimum number of stages and steps) of a *single* library sequence ('goal-part'), with worst case O(n^3) scaling where *n* is the number of subparts, and where the minimum number of stages is ceil($\log_2(n)$). They extended this to multiple-goal-parts by introducing 'slack' and 'sharing' factors that assemble all goal-parts by sharing nodes between trees to create a graph which reduces the number of steps, with O($k^2 n_{\max}^2$) scaling where *k* is the number of single goal-part assembly graphs created, $n_{\max}$ is the maximum number of subparts across all goal-parts, and the minimum number of stages is ceil($\log_2(n_{\max})$). In what follows we refer to this algorithm as A_1, and our new algorithm as A_{new}.

Greedy algorithms apply locally optimal choices at each stage to approximate, or even obtain, a globally optimal solution in reasonable time; ideally for problems exhibiting 'optimal substructure' (also necessary for dynamic programming).

2. ALGORITHM

Unlike [1] we address libraries of many sequences directly. Our algorithm leverages the fact that all concatenation steps occur between a *pair* of adjacent parts (ordered 2-tuples of the *p*-tuples described in section 2.1). Therefore, when applied greedily to all sequences, each pair is mutually exclusive of any other pairs, either to the left or right of an occurrence that contain the 1st or 2nd part respectively; i.e. overlapping pairs constituting a 3-tuple *triple* which cannot both be concatenated in the same stage.

We rank concatenations using a *scoring* function that is simply the number of occurrences of that pair throughout the library, although other biological or manufacturing constraints could be taken into account by alternative scoring functions.

2.1 DNA Library Encoding and Example

Sequences of the DNA library to assemble are encoded in terms of their constituent subparts, i.e. as a list of *s*-tuples (or any hashable ordered sequence) of *p*-tuples of hashable objects representing subparts. Table 1 shows the state of the four sequence example library as it evolves under the optimal assembly plan shown in Figure 1 (sequences are shown as hashable Python objects conforming to this library encoding).

Table 1: Library state and resulting steps in a run of A_{new}.

Initial input	After stage 1	After stage 2

`library=[`
`((A,),(B,),(E,)),`
`((A,),(B,),(D,),(E,)),`
`((A,),(C,),(D,),(E,)),`
`((A,),(D,),(E,))]`
	`library=[`
	`((A,B),(E,)),`
	`((A,B),(D,E)),`
	`((A,C),(D,E)),`
	`((A,),(D,E))]`
		`library=[`
		`((A,B,E)),`
		`((A,B,D,E)),`
		`((A,C,D,E)),`
		`((A,D,E))]`

`steps=[`
`((A,),(B,)),`
`((D,),(E,)),`
`((A,),(C,))]`
	`steps=[`
	`((A,B),(E,)),`
	`((A,B),(D,E)),`
	`((A,C),(D,E)),`
	`((A,),(D,E))]`
		`steps=[]`

Note that since optimal steps were derived in stage 1, the `steps` of stage 2 are the same as the sequences in `library` after stage 1.

2.2 Description

For each stage, *maps* from pairs and triples to *multisets*, counting the number of times they occur in a sequence, are computed and from these, *multimaps* of pairs to containing triples and triples to contained pairs, are derived.

Using the per-pair scoring function, a list of (pair, score) items is computed, shuffled (to randomize the order of equal scoring pairs in a stable sort - allowing multiple runs with different resultant assembly graphs) and sorted by descending score.

For each (pair, score) a pair is added to a list of *excluded* pairs if it has a score ≤ 0 (allowing custom scoring functions to exclude certain pairs by design) or, if not already excluded, it is added to the set of steps for this stage and all other pairs that are in a triple with the current pair are excluded instead. If no steps were added in this stage then the list of stages, containing the lists of steps for each stage, is returned.

Lastly, each of the highest scoring pairs that are not excluded by a higher scoring pair are then concatenated by seeking for occurrences in the sequences of the multiset mapped to that pair and moving elements of the right p_r -tuple into the left p_l-tuple, leaving a pair of ($p_l + p_r$)- and 0-tuples, so as to not create new occurrences of any pairs.

Once all possible concatenations have been applied, all 0-tuples are removed and the algorithm loops.

3. RESULTS

We compare the performance of our algorithm to that of A_1 in terms of assembly graph quality (smallest number of stages then steps, Table 2) and running time (Table 3) using the one synthetic and two real world libraries of [1]: a small dataset for exhaustive search, 'phagemid' and 'iGEM-2008' (containing 2 duplicates).

Table 2: Quality (|stages|,|steps|) of assembly graph

Library |size|	Exhaustive	Phagemid	iGEM-2008
Algorithm	**|5|**	**|131|**	**|395|**
A_1	(3,11)	(4,202*)	(5, 808*)
A_{new}	(3,11) best (3,15) worst	(4,202) or (4,208) †	(5,808) best (6,841) worst

* A_1 values were estimated from [1] Figure 7. † Only outcomes.

Table 2 shows that our algorithm A_{new} obtains the optimal solutions found by A_1. Therefore we conclude that this greedy approach can optimally assemble real-world DNA libraries.

Table 3: Running time (seconds) for subsets of iGEM-2008

Library size	50	100	150	200	250	300	350	395
Algorithm								
A_1 ‡	0.1	1	45	120	290	405	620	720
A_{new} mean of 10 runs	$.007 \pm 10^{-5}$	$.014 \pm 10^{-4}$	$.020 \pm 10^{-5}$	$.027 \pm 10^{-4}$	$.033 \pm 10^{-4}$	$.039 \pm 10^{-4}$	$.045 \pm 10^{-4}$	$.053 \pm 10^{-4}$

‡ A_1 values were estimated from [1] Supplemental Figure 1.

Table 3 shows the running times of A_{new} were at least two orders of magnitude faster than A_1 and imply that A_{new} scales linearly with the number of sequences. The implementation language, runtime and hardware these timings were obtained with may differ; we used Python running on a single i7-2670 2.2Mhz core.

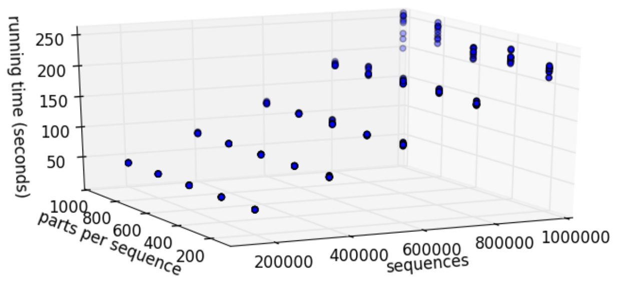

Figure 2: Running times of generated large-scale combinatorial DNA libraries for A_{new} (10 runs each).

To investigate the growth of running time with library size further, we generated synthetic combinatorial libraries with up to 1 million sequences and 1000 parts per sequence. Figure 2 shows how the running times on these synthetic libraries continue the trend observed in Table 3: increasing linearly with the number of sequences; and that |parts| per sequence has only a minor effect.

4. PROJECTS

This work was supported by CADMAD (FP7 STREP 265505), ROADBLOCK (EP/I031642/1) and AUDACIOUS (EP/J004111/1).

5. REFERENCES

[1] 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 Research 38, 8 (Mar. 2010), 2607-2616. DOI=http://dx.doi.org/10.1093/nar/gkq165