Optimizing Combinatorial Library Construction via Split Synthesis

Barry Cohen
Steven Skiena
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

1 Introduction

Combinatorial chemistry is one of the most exciting recent developments in drug discovery, the foundations of which rest on the systematic synthesis and screening of large libraries of small molecules. In the standard one-bead, one-compound approach [11] to combinatorial chemistry, a large number of distinct small molecules are fabricated on distinct resin beads. These beads are reacted against a target to establish which beads contain biologically active agents. The compound associated with a positively-reacting bead can then be identified via sequencing or indirect coding methods.

The adjective "combinatorial" in "combinatorial chemistry" refers to the large number of distinct molecules (on the order of 10^6 to 10^7) which can simultaneously be assayed. Unfortunately, the combinatorial explosion ensures that desired screening libraries quickly exceed this bound. For example, full libraries of all l-peptides cannot be fabricated for $l \geq 6$, since there are 20^l distinct molecules containing l amino acid residues. In response, techniques for constructing representatives for all of a simple set of motifs have been developed using random synthesis steps [11]. However, the sophistication and hence effectiveness of such libraries is limited by the difficulty of designing cost-effective synthesis strategies.

In this paper, we propose a new algorithmicly-intensive *optimized split-synthesis* technology for fabricating combinatorial chemistry libraries and evaluate its potential through extensive simulation. We believe our methods hold promise in synthesizing libraries which are (1) too large to reasonably employ sequential synthesizers (say greater than 50 distinct molecules), (2) too long or irregular for conventional split-synthesis generation techniques [11], and (3) not used in production quantities—thus making array-based synthesis techniques [4] economically unviable.

Split synthesis, developed independently by Furka [5] and Lam [9], is the method of choice to build libraries for combinatorial chemistry. The basis of this technology is that molecules can be grown with one end tethered to a resin bead, while the other end is extended one residue at time in parallel across a set of beads.

Library construction proceeds through an interleaved sequence of divide, grow, and combine operations. A set of beads can be *divided* or partitioned into different chambers, and all the molecules on the beads in a particular chamber *grown* or extended by the same set of residues. After this reaction, the sets of beads can be *combined* or mixed together so they can be grown within the same chamber. Once combined, two sets of beads cannot be separated again, although the bead mixture can be divided at will. Figure 1 illustrates a procedure to synthesize a library of the oligonucleotides $\{AATT, AGGA, ATTT, GAAT, GAGG, GTCC, TATT, TGGA, TTAA, TTGT\}$.

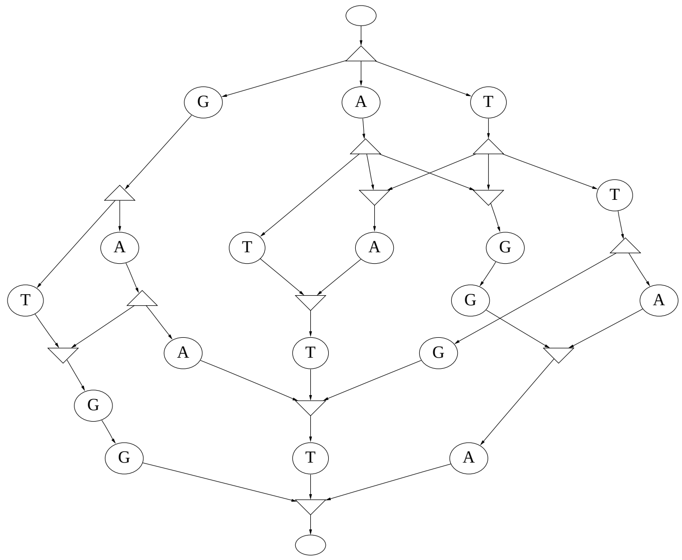

Figure 1: A procedure synthesizing a given 10-string library.

It is straightforward to fabricate all 20^l l-peptides (over the 20 amino acids) or 4^l oligonucleotides (over the bases A, C, G, and T) using divide, grow, and combine operations. Partition the initial beads into 4 or 20 chambers, grow each by a different residue, combine all the beads, and repeat for a total of l-iterations. Much more complicated is fabricating beads for each instance of a class of possible patterns or *motifs*. A typical pattern of interest might be all l-peptides with certain critical residues in fixed positions. For sufficiently complex motifs, an efficient synthesis schedule is difficult or impossible to design by hand.

In this paper, we present techniques for constructing efficient split-synthesis protocols which fabricate a desired target library. We seek to minimize the number of *grow* operations performed over the course of the synthesis, since these most accurately reflect the cost and complexity of the protocol. Note, however, that we can construct a superset of the specified library if it will simplify the synthesis, provided the total number of compounds synthesized doesn't grow too large. Thus optimal protocol design involves making the right tradeoff between these two criteria.

Our paper is organized as follows. In Section 2, we present our model for split-synthesis library design, and develop the associated algorithmic theory, demonstrating that it is NP-complete to optimize library construction even over an extremely restricted class of libraries. Section 3 defines the high-level operations on a dag which we employ to optimize the library, and the elementary operation from which the high-level operations are constructed. Sections 4 and 5 present two distinct and complementary computational approaches to library construction, bottom-up and top-down. These approaches prove to be closely connected to the problem of partitioning graphs into bipartite cliques, and we include a technical result on an interesting special case of clique partition in the Appendix. Experimental results are reported in Section 6. Section 7 discusses our system architecture and associated implementation issues. Section 8 presents our conclusions concerning the sizes and types of libraries which our techniques hold promise to build, as well as open problems.

Our efforts to improve split-synthesis library techniques are a natural follow up to our previous work with a related technology: constructing dense oligonucleotide arrays on the Southern Array Maker apparatus for interactive sequencing by hybridization, more fully described in [2, 13].

2 Modeling Split Synthesis

We seek to develop and implement algorithms which take as input a set of desired peptides or oligonucleotides, and output the split-synthesis schedule of an efficient series of reactions to fabricate them.

Split synthesis can be modeled by a directed acyclic graph G, in which each node represents a grow step on a particular subset of beads. The node is labeled with the unit which is appended by the grow operation. The combine and divide steps must be represented explicitly in a production schedule, but since they can be easily reconstructed from G, we will discuss only the implicit dag representation. If v_i is a node of G, then each path from the source of G to v_i represents a particular compound; the set of all such paths corresponds to the set of commingled beads at v_i.

Formally, then, the *minimum split synthesis construction problem* is:

Input: A set S of n strings of length k over a finite alphabet Σ, and integers $R \geq n$ and m.

Output: Does there exist a vertex-labeled dag G with at most m nodes such that:

1. Each $s \in S$ is represented by at least one path in G
2. The total number of paths through G is $\leq R$.

It is important to note that our problem is not the same as minimizing a DFA—a well-studied problem for which efficient algorithms are known [7]. The minimum dag which solves our problem may be "nondeterministic," that is, from a given node it has edges to identically labeled children. Figure 2a gives an example in which the smallest deterministic construction realizes $\{aba, bba, bbc, cbc\}$ with 8 nodes, while the optimal dag (Figure 2b) realizes the same strings with 7 nodes. Unfortunately, the problem of optimizing synthesis proves to be hard even in highly restrictive conditions:

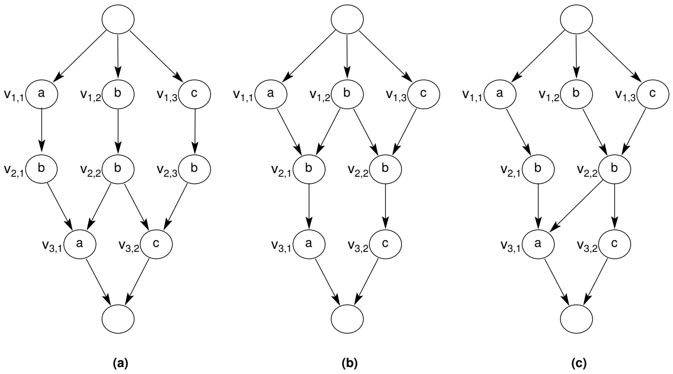

Figure 2: (a) The smallest deterministic dag synthesizing the strings $\{aba, bba, bbc, cbc\}$. (b) The smallest unrestricted dag which realizes the same set of strings. (c) The dag resulting from the merger of nodes $v_{2,2}$ and $v_{2,3}$ in (a), which also realizes string *cba*.

Theorem 1 *The minimum split synthesis construction problem is NP-complete, even for the special case in which all strings are of length 3, all strings have the same second character, and no strings distinct from the target strings may be added.*

Proof: Clearly the problem is in NP. To show hardness, we use a reduction from "covering by complete bipartite subgraphs" [6], which which asks whether there is a cover of the edges of a bipartite graph G by k complete bipartite subgraphs (cliques). All the edges of each clique must appear in G and each edge in G must appear in at least one clique. (An edge or vertex of G may appear in more than one clique.)

The reduction is as follows. Suppose $\{X, Y\}$ is a bipartition of the vertices of G. For each edge (x_i, y_j) in G, construct a string $x_i A y_j$. We show that there is a k cover of G iff there is a dag G' with $k' = |X| + |Y| + k$ vertices constructing these strings.

Clearly, each x must occur on level 1 of G', and each y on level 3. Further, only one vertex with each label is needed on each level. If two vertices with the same label exist on the same level, we merge them. The set of strings produced is unaffected. Since $|level\ 1| = |X|$ and $|level\ 3| = |Y|$, we really seek to determine if there exists a G' with k A-vertices on level 2.

Note that all the paths in G' corresponding to a clique in G may be constructed by a single A vertex. Likewise, each pair of above and below neighbors of a vertex A must be an edge in G. Hence each A defines a bipartite clique in G. Now there exists a k covering of G if there is a G' with k A-vertices. Further, a k covering of G naturally defines a G' with k A-vertices which construct our target strings. Hence there is a k covering of G iff there exists a G' with k' vertices. ∎

Thus optimizing split-synthesis automata is intimately related to covering the edges of a graph by complete bipartite subgraphs (bipartite cliques) [6]. Given a fixed graph

G, and a class of graphs H, the minimum number of H-subgraphs which cover the edges of G is denoted $\mathrm{cov}(G, H)$. Results on bounding this minimum value for various H are surveyed by Pyber [3, 14]. A related literature exists in the field of communications complexity, which asks the minimum amount of communication required for two parties to share information. The communications complexity of a graph G is the minimum number of bipartite cliques to cover all its edges, and is bounded between $\log rank(G)$ and $rank(G)$. See Lovasz [12] for details.

In light of the application to split synthesis and the hardness of partitioning bipartite graphs into bipartite cliques, it is interesting to establish how much one must restrict the permitted degree of the vertices to make the problem tractable. In the Appendix, we show that if vertices of degree 3 or larger are permitted, the problem becomes intractable, although for degree 2 the problem remains open.

3 Local Optimization on Vertex-Labeled Dags

In this paper, we will develop two distinct but complementary approaches to experimental design. In each, we construct an initial dag G, containing all the desired strings. Then we apply a greedy heuristic to identify a sequence of local operations on nodes and edges, each of which brings us closer to the minimal solution, while ensuring that all desired paths remain. Our two approaches, *bottom-up* and *top-down*, differ in how the initial dag is constructed, and which local operations are employed.

We define all our local modifications of G in terms of a single underlying operation, which we call *string transfer*. The string transfer operation guarantees that a path corresponding to each input string is preserved.

Let $s_{i,j}$ be the jth letter of the ith input string, and $v_{m,n}$ be the nth vertex of G on level m, where level denotes the shortest-path distance between the source and the node. We maintain a mapping P from each $s_{i,j}$ to some $v_{j,k}$ whose label is the character $s_{i,j}$. That is, the labels on the successive values of P form a source-to-sink path in G corresponding to the input string s_i. In Figure 3a, the dashed paths indicate the values of P for the set of input strings $\{aba, bba, bbc, cbb\}$. For example, string bba passes through $(v_{0,1}, v_{1,2}, v_{2,1}, v_{3,1}, v_{4,1})$. Conversely, each dag node has a set of strings passing through it. The set of strings passing through $v_{2,1}$ is $\{aba, bba\}$.

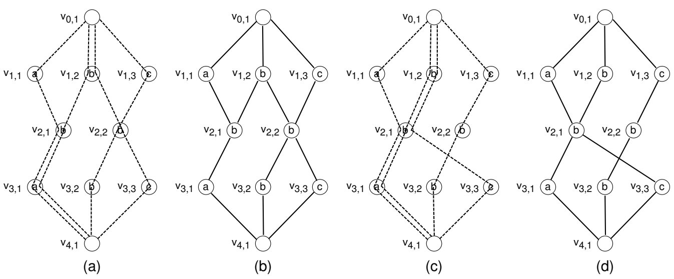

Figure 3: (a) The dashed lines represent the mapping P from the letters of four input strings, $\{aba, bba, bbc, cbb\}$. (b) The dag determined by (a). (c) The mapping P after string bbc has been remapped (transferred) from $v_{2,2}$ to $v_{2,1}$. (d) The dag determined by (c).

P determines G in a natural way, guaranteeing that every input string is fabricated. Each $P(s_{i,j})$ is a node in G. There is an edge in G between two nodes on adjacent levels iff some string s_i passes through them both. Now, for every two consecutive letters of the input, $s_{i,j}$ and $s_{i,j+1}$, $P(s_{i,j})$ and $P(s_{i,j+1})$ are neighbors in G. Therefore, s_i is fabricated by the sequence of vertices $P(s_{i,1}), \ldots, P(s_{i,k})$, where k is the length of s_i. Figure 3b represents the dag induced by the paths of Figure 3a.

Say the path associated with s_i is $(p_{i,1}, \ldots, p_{i,j-1}, x, p_{i,j+1}, \ldots, p_{i,k})$. We can transfer s_i from node x to node y iff y has the same label as x and is also on level j. Edges $(p_{i,j-1}, y)$ and/or $(y, p_{i,j+1})$ must be added to G unless they have previously been defined by some other path. Similarly, edges $(p_{i,j-1}, x)$ and $(x, p_{i,j+1})$ can be deleted from G if they are not defined by any other path. If no string passes through x after the transfer, x may be deleted from G. Figure 3c illustrates the transfer of the second letter of string bbc in Figure 3a from $v_{2,2}$ to $v_{2,1}$. Figure 3d shows the dag which results from this transfer.

A transfer of $s_{i,j}$ from node x to y defines a set of *induced strings* whose transfer to y is certain not to require any additional edges in G and which therefore can not increase the number of paths through G, but which may decrease the number of paths and/or nodes in G. Input string s_k is induced if edges $p_{k,j-1}, y$ and $y, p_{k,j+1}$ already exist. In Figure 3, any string with a b on level 2 and which passes through $v_{1,1}$ or $v_{i,2}$ on level 1 and through $v_{3,1}$ or $v_{3,3}$ on level 3 could be transferred to $v_{2,1}$ without adding any edges to G. Therefore, when performing a string transfer, we may transfer any or all induced strings as well.

Three local operations are readily defined in terms of string transfer:

- *Merge nodes*—Any two identically-labeled nodes x and y on level l can be merged by transferring all strings passing through y to x. We reduce the size of the dag by one vertex, possibly at the cost of adding new paths formed by concatenating source-to-x (source-to-y) subpaths and y-to-sink (x-to-sink) subpaths.

- *Split a node*—We can split any node x by creating a new node y with the same label as x, and transferring a proper subset of the strings passing through x to y. We increase the size of the dag by one vertex, while potentially reducing the number of non-input paths. Although two nodes can be merged in only one way, node x can be split by any partition of the strings passing through it into two nonempty subsets.

- *Refine a node*—To refine node x on level i, we delete an edge (x, y) from G by transferring each string passing through both x and y from x to a different vertex on level i. This may require the addition of one or more edges in G.

  The string transfer in Figure 3 is an example of refinement. Let node $v_{2,2}$ represent x, and node $v_{1,2}$ its neighbor y. Since only string bbc passes through both y and x, this refinement results in the deletion of the edge $(v_{1,2}, v_{2,2})$ at the cost of adding edge $(v_{2,1}, v_{3,3})$. It pays to do such a refinement only if the total number of paths through G is thereby reduced. In this example, the two paths bbb and cbc are eliminated at the cost of adding abc, so this is a desirable transfer. If all strings are refined from node x, then x may be deleted as well.

4 Bottom-Up Experimental Design

In this section, we consider a bottom-up approach to optimizing split-synthesis. It starts with an initial dag which realizes exactly the target set of strings, and then performs local operations that reduce the number of nodes at the cost of possibly adding strings.

4.1 Initial Dag Construction

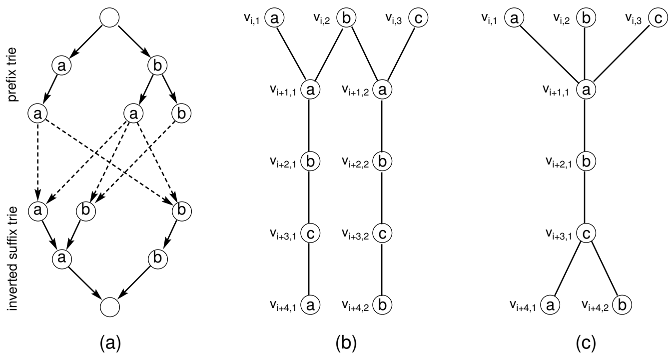

Figure 4: (a) The initial bottom-up construction of a dag for strings $\{aaaa, aaba, baaa, baba, babb, bbba\}$. (b) The simple paths consisting of nodes $(v_{i+1,1}, v_{i+2,1}, v_{i+3,1})$ and $(v_{i+1,2}, v_{i+2,2}, v_{i+3,2})$ are candidates for merging. (c) The dag after the simple paths have been merged.

The initial dag for our bottom-up strategy is the conjunction of two tries. One trie is built from the prefixes of S and the other (inverted) trie is built from the suffixes of S. (Figure 4a.) We divide each input string of length k into a prefix of length $k/2$ and a suffix of length $k/2$. The prefixes are represented by a conventional prefix trie. Level 0 of the trie consists of the source, and on level i at most α^i nodes, one for each distinct prefix of length i.

The suffixes are similarly represented by an inverted trie. The sink will be at level $k+1$, and there is one leaf for each distinct suffix at level $k/2$. In the suffix trie we reverse the direction of the edges, so that edges point away from the leaves towards the sink.

For each input string $s \in S$ we create an edge between the node corresponding to the last letter of its prefix and the node corresponding to the first letter of its suffix. The result is a dag with a single source and a single sink. Moreover, for each input string there is a unique path through the dag and every path corresponds to an input string.

We can now perform two classes of optimizations. *Fixed library* optimizations alter the dag, but not the set of labeled paths through it. *Enhanced library* optimizations alter the dag, preserving all input paths while adding and/or deleting other paths.

4.2 Fixed Library Optimizations

Note that if two nodes, x and y have the same label and an identical set of parents, they can be merged without altering the set of labeled paths through the dag. This *pairwise reduction* can also be applied to any two nodes with a common set of children, since there is a symmetry between parents and children.

Identifying the best series of pairwise reductions is complicated by the fact that a given node may have all its parents in common with one node and all its children in common with another. In our programs, we simply perform pairwise reductions in the order they are discovered. The only candidates for pairwise reduction with node x are the children of x's parents or the parents of x's children. Since G is generally sparse, searching this list of candidates is an efficient operation which may be applied periodically. In our experiments, pairwise reductions account for between 2% and 33% of the total reductions achieved over the initial dag, depending on the input.

4.3 Enhanced Library Optimizations

We augment the previously discussed node merger and node refinement with a more powerful *path merger* optimization.

We call a path in G *simple* if exactly the same set of strings pass through all its vertices. A path in G is simple iff every node after its head node has only one parent and every node before its tail node has only one child. Each node in G is a simple path of length one.

Let A and B be two identically labeled simple paths in G of length L occurring the same distance from the source. Figure 4b shows an example in which path A consists of nodes $(v_{i+1,1},\ v_{i+2,1},\ v_{i+3,1})$ and path B consists of $(v_{i+1,2},\ v_{i+2,2},\ v_{i+3,2})$.

We may merge A and B by transferring all strings passing through B into the corresponding nodes of A. All the nodes of B are deleted. (Figure 4c.)

Path merger preserves all existing labeled paths through G, and reduces the number of nodes in G by the length of the paths (the benefit) while possibly adding new paths through G (the cost). In Figure 4, three nodes are deleted and two paths are added.

The merger of two nodes is a special case of path merger in which the paths are of length one.

In our implementation, we apply path merging in strictly greedy fashion. All combinations of cost and benefit are ranked, and the cheapest are performed first. Mergers are performed until the cumulative cost in strings equals the maximum price we are willing to pay.

After optimizing G using this merge strategy, refinement typically reduces the number of paths through G by an additional 15-25%.

5 Top-Down Experimental Design

Our top-down technique for split-synthesis design starts with the smallest possible initial dag which realizes the target library. It realizes a large fraction (perhaps all) possible length-k strings over the given alphabet. We then perform local operations to reduce the number of undesired strings while adding a small number of additional nodes. The potential advantage of this approach is that our initial dag comprises at most $k\Sigma$ nodes, so we can afford a large number of splits which reduce unneeded strings.

Figure 5a illustrates the initial dag construction. We start with either zero or one node of each given label on each level of G, depending on whether the given label appears at that position in the input. On a large input, this may yield a level-wise complete dag, in which each node is a neighbor of every node on neighboring levels. Such a complete dag generates Σ^k paths, most of which are likely to be unnecessary.

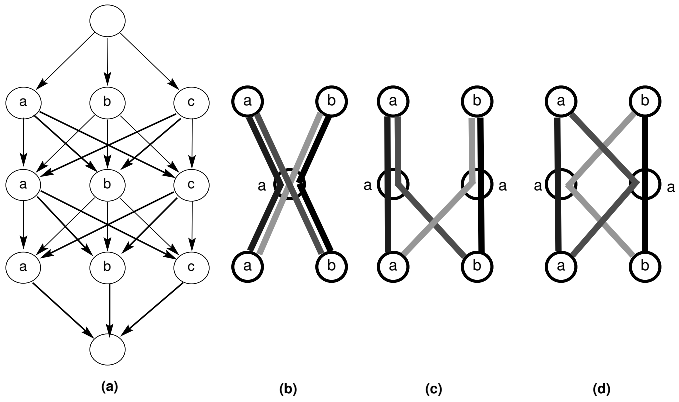

Figure 5: (a) The initial construction of a top-down dag. Any split of the middle node of (b) results in the same (c) or a greater (d) number of paths.

We seek to split nodes to reduce the number of paths through G. Unfortunately, after our initial construction it is often the case that no node can be split productively, as in Figure 5b. Every split produces the same number of paths (Figure 5c) or a greater number (Figure 5d). A more powerful technique is required to reduce the number of paths.

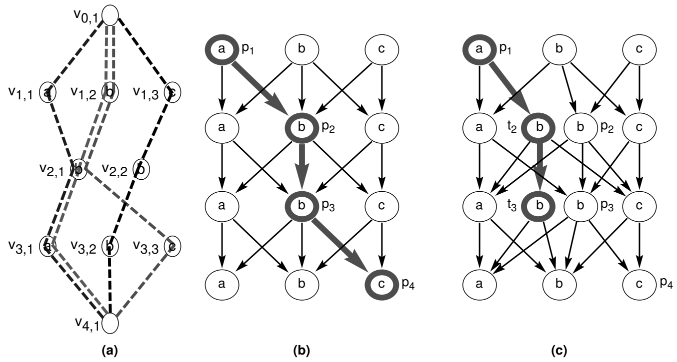

Figure 6: (a) A path is forbidden if no string passes through all its vertices. Path $(v_{1,1},\ v_{2,1},\ v_{3,3})$ is forbidden. (b) Assume $p = (p_1,\ p_2,\ p_3,\ p_4)$ is forbidden. Construction (c) eliminates all paths through G containing p as a subpath.

Every path p in G of length $j \geq 3$ either has some string $s \in S$ passing through all nodes of p, or it does not. If so, we say p is a *required* path; otherwise, p is a *forbidden* path. All paths of length 1 or 2 are required, since by definition a node exists in G only if a string passes through it, and an edge exists between two nodes on neighboring levels in G iff some string passes through both of them.

If p is a forbidden path, we can eliminate all paths in G which contain p as a subpath. The construction is as follows:

- Create a twin for each internal node of p. Call them $t_2 \ldots t_{j-1}$. Initially, each twin is empty, i.e., no string passes through it.

- Now consider each string s which passes through p_1, the head of p. Say s passes through the first l nodes of p. For $2 \leq j \leq l$, transfer s_j from p_j to t_j.

Lemma 1 *Splitting forbidden path p eliminates all paths in G which contain p as a subpath, while maintaining paths representing all input strings.*

Proof: Figure 6 shows how path splitting works for a subgraph of 12 nodes in which the path p is forbidden, i.e., in which no string passes through nodes (p_1, p_2, p_3, p_4).

Consider the impact of splitting path p. Clearly, any string which does not pass through p_1 is unaffected. So let us concentrate on what happens to the strings which do pass through p_1.

In the first step of our construction, twin nodes t_2 and t_3 are created. If a string s only passes through $p_1 \in p$, it is unaffected. If s passes through p_1 and p_2, it is transferred from p_2 to t_2, and from there continues as before. If s passes through p_1, p_2 and p_3, it is transferred from p_2 to t_2 and from p_3 to t_3, and from there continues as before. By assumption, no string passes through all four nodes of p, or else p would be a required path. Consequently, after the split there is no edge in G between t_3 and p_4.

We have succeeded in eliminating from G all paths which contained p as a subpath—none of which was needed to witness the construction of the input strings. ∎

The number of paths eliminated is the number of paths from the source to p_1 times the number of paths from p_j to the sink. This is the *primary benefit* of splitting p. There may be additional benefits. If a forbidden path q is formed by any prefix of p plus one additional node as its tail, splitting p in this way also eliminates all paths in G which contain q as a subpath.

Splitting path p adds $|p| - 2$ nodes to G. When $|p| = 3$, a single node is split, forming a $K_{1,x}$ clique.

We apply the path-splitting strategy according to a greedy criterion. All forbidden paths less than a critical length are located, their cost/benefit ratio calculated, and the best split performed. At specified intervals, pairwise reductions are performed. G is refined at the conclusion of path splitting.

We note that the procedure may terminate when all forbidden paths up to a specified length have been eliminated.

6 Experimental Results

To compare the performance of our two heuristics and assess the range over which our techniques are practical, we performed extensive experiments on the following data sets:

- All k-mers of the viral DNA sequence SIV for $k = 10$, 12, 14, 16, 18 and 20. This data set is intended to be representative of libraries designed to combat a specific target organism.

- The language RE of the regular expression $((a+b+c)(a+b))^5$, containing 7,776 strings of length 10. This data set represents libraries designed around highly structured motifs.

- The languages *REMINUS* and *REPLUS*, created by deleting/adding 256 random strings to *RE*. These data sets are intended to represent minor variations of highly structured libraries.

- All k-peptides of the 500-peptide protein FTF-HUMAN and the 1,000-peptide protein MYSH-CHICK for $k =$ 5, 6, 7, 8, 9 and 10. These data sets evaluate our techniques for the 20-character amino-acid alphabet associated with proteins.

- The languages RND-10, RND-12, RND-14, RND-16, RND-18 and RND-20, each consisting of 256 randomly generated words of the prescribed length over the alphabet $\{A, C, G, T\}$. These data sets evaluate our techniques on smaller libraries than presented above.

The most trivial approach to library construction requires a number of grow steps equal to the total number of letters in the input. This count provides one baseline for evaluating our results. A fairer baseline is the size of the minimal directed acyclic word graph we use as the initial dag in the bottom-up method. In our tables below, we present both of these baselines, plus the number of nodes in our final optimized dag for a given R, the maximum number of acceptable strings.

Table 1 and Figure 7 present the results for optimization of the SIV data sets. The initial bottom-up dag construction produces substantial savings over the input size, ranging from a factor of 21.7 on SIV-10 to 2.3 on SIV-20. Columns 5 and 7 give the number of nodes for the bottom-up and top-down methods when $R = 100,000$. After optimization, the top-down method has reduced grow steps compared to initial input size by a factor of 80.6 for SIV-10. The bottom-up method has reduced grow steps compared to initial input size by a factor of 5.6 for SIV-20.

Figure 7 provides a way to compare the effectiveness of the top-down and bottom-up approaches to optimization. Each point on each plot shows a realized tradeoff between nodes and paths. The horizontal (for bottom-up) and vertical (for top-down) tails on these trajectories denote effects of a final refinement operation to clean up the dags. The effect of this refinement can be significant—it reduces the string count by more than 80% for certain inputs.

Using the top-down approach for SIV-10, 634 nodes are required to reduce the string count below 100,000, a reduction by a factor of 80.6 from the initial input size; the comparable figure for bottom-up is 968. For SIV-12, 14, 16, 18 and 20 the bottom-up approach requires slightly fewer nodes than the top-down approach.

Table 2 and Figure 8 present the results for substrings of the 1,000-peptide protein MYSH-CHICK. The bottom-up method produces slightly better results on all length substrings.

Table 3 and Figure 9 present similar results for the smaller 500-peptide protein FTF-HUMAN. Again, the bottom-up method produces slightly better results on all length substrings. Grow steps are reduced by a factor of 7.7 for FTF-05 and 2.6 for FTF-10. These protein results are quite encouraging, showing that interesting protein libraries can be synthesized using our methods. For comparison, note that the complete set of $20^5 = 3,200,000$ 5-peptides is already infeasible to construct using conventional split-synthesis technology.

The language RE defined by $((a + b + c)(a + b))^5$ can be synthesized using 25 grow steps; both optimization techniques arrive at this result. REMINUS, created by deleting 256 strings from this regular expression, can be efficiently synthesized using 25 steps by adding back the 256 deleted strings. REPLUS is more difficult; the trivial construction of the 256 added strings alone requires 2,560 nodes. Figure 10 and Table 4 present the results of the bottom-up technique on these languages. The resulting dag constructs REMINUS with 99 nodes, while adding fewer than 256 strings. The top-down approach initially constructs exactly the 25-node automaton. The bottom-up approach constructs REPLUS with only 404 nodes for $R = 15,000$; the split approach employs 456 nodes.

Finally, we consider the construction of the family of smaller random languages. Figure 11 and Table 5 present the results. Keeping $R \leq 50,000$ in every case, the improvement over the linear construction ranges from 10.9 (the top-down method on RND-10) to 3.3 (bottom-up on RND-20).

7 System Architecture

Our simulation is implemented in C++, and the data presented here were generated on a 300 megahertz PC with 192 MB of memory. All results presented here for the bottom-up technique were computed in under fifteen minutes. The top-down approach is more time consuming, but takes under eight hours on each run. We believe that the run times for the second method could be improved significantly were it important to do so.

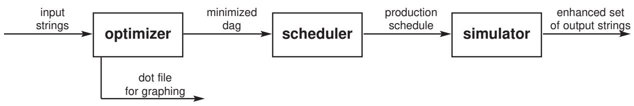

Figure 12: System Architecture. Accepts set of target strings, and outputs an optimized laboratory procedure for synthesizing a superset of the input.

Our implementation consists of the following components:

- The *optimizer* accepts a target set of strings over an arbitrary alphabet. A typical alphabet might range from size 4 (nucleotides) to 20 (amino acids). The size of the input, i.e. the number of input strings times their length, is limited by available memory. One million characters is feasible on a machine with 384 megabytes of memory. The optimizer also takes as input a desired number of strings in the output.

  The optimizer outputs an abstract representation of the synthesis process in the form of a dag, various statistics on the dag and also a dot file [8] which is suitable for displaying pictures of small dags.

- The *scheduler* takes a dagfile as input, and various parameters of the laboratory setting. Most important is the number of grow operations which can be performed in parallel. The scheduler outputs a detailed specification of the laboratory procedure.

- The *simulator* emulates the specified laboratory procedure and outputs the set of strings it produces.

8 Conclusions

These results indicate that our combinatorial approach to split synthesis can achieve significant improvements across a range of input sizes and on varied alphabets. Both methods may be effectively applied both to DNA/RNA synthesis with an alphabet of four, and to protein synthesis with an alphabet of 20.

The efficiencies achieved are greatest in synthesizing languages with strong regularities, but which are not simple enough to be designed by hand. This suggests our methods may best be applied when the object is a set of variations on one or more motifs. The "excess" strings produced, which

input	string count	raw input size	Bottom-Up Optimization initial size	final size	Top-Down Optimization initial size	final size
SIV-10	5,107	51,070	2,351	968	42	634
SIV-12	5,155	61,860	6,957	2,480	50	2,795
SIV-14	5,163	72,282	14,608	5,121	58	7,020
SIV-16	5,165	82,640	23,950	8,691	66	9,884
SIV-18	5,167	93,006	33,922	12,292	74	13,601
SIV-20	5,169	103,380	44,106	16,890	82	18,381

Table 1: Optimization of SIV k-mers

input	string count	raw input size	Bottom-Up Optimization initial size	final size	Top-Down Optimization initial size	final size
MYSH-05	996	4,980	1,576	503	102	629
MYSH-06	995	5,970	2,468	906	122	1,060
MYSH-07	994	6,958	3,454	1,411	142	1,531
MYSH-08	993	7,944	4,438	1,918	162	2,090
MYSH-09	992	8,928	5,426	2,531	182	2,639
MYSH-10	991	9,910	6,413	3,180	202	3,190

Table 2: Bottom up and top down optimization of MYSH-CHICK, for $R = 20,000$

input	string count	raw input size	Bottom-Up Optimization initial size	final size	Top-Down Optimization initial size	final size
FTF-05	496	2,480	1,044	324	102	361
FTF-06	495	2,970	1,515	515	122	688
FTF-07	494	3,458	2,005	824	142	965
FTF-08	493	3,944	2,491	1,108	162	1,283
FTF-09	492	4,428	2,981	1,426	182	1,597
FTF-10	491	4,910	3,466	1,748	202	1,915

Table 3: Bottom up and top down optimization of protein FTF-HUMAN for $R = 20,000$.

input	string count	raw input size	Bottom-Up Optimization final nodes	strings added	Top-Down Optimization final nodes	strings added
RE	7,776	77,760	25	0	25	0
REMINUS	7,520	75,200	99	234	25	0
REPLUS	8,032	80,320	404	5,505	524	1,703

Table 4: Optimization of RE, REPLUS and REMINUS

input	string count	raw input size	Bottom-Up Optimization initial size	final size	Top-Down Optimization initial size	final size
RND-10	256	2,560	955	353	40	234
RND-12	256	3,072	1,419	530	48	430
RND-14	256	3,584	1,962	753	56	602
RND-16	256	4,096	2,477	998	64	808
RND-18	256	4,608	2,981	1,243	72	1,082
RND-20	256	5,120	3,486	1,543	80	1 578

Table 5: Synthesis of random strings over $\{A, C, G, T\}$, with $R = 50,000$.

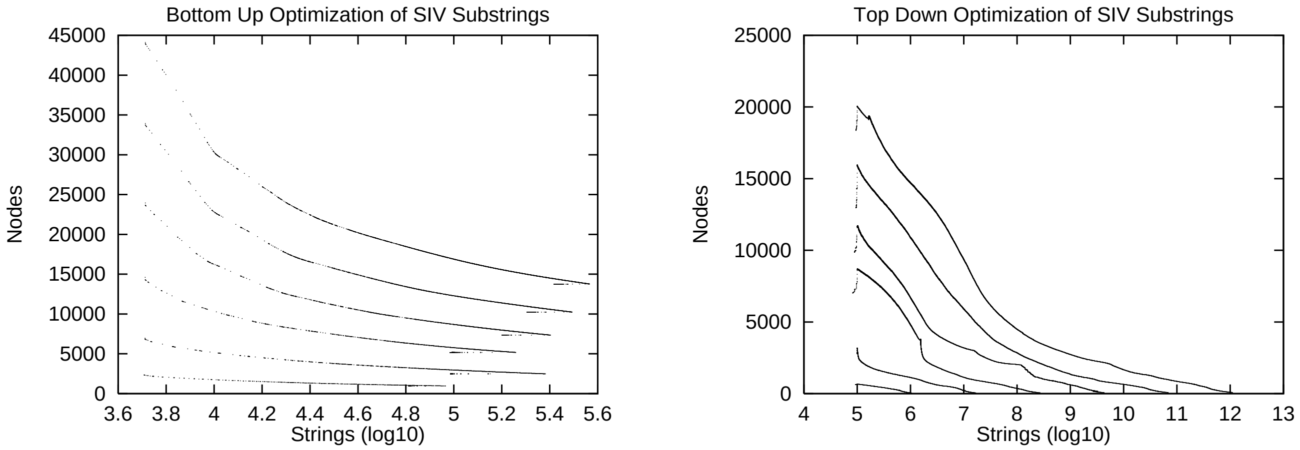

Figure 7: Bottom up and top down synthesis of all k-mers of the virus SIV, $k = 20$, 18, 16, 14, 12, and 10.

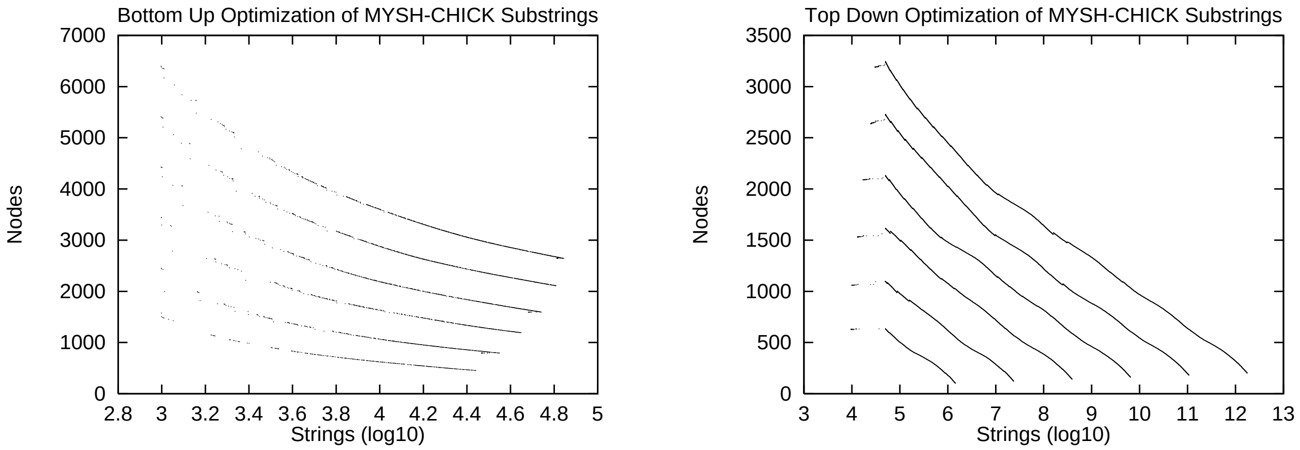

Figure 8: Bottom up and top down synthesis of all k-peptides of MYSH-CHICK, $k = 10$, 9, 8, 7, 6, and 5.

we have thus far described only as an inefficiency, are themselves potentially interesting variations on the target input strings, and hence potentially valuable for drug discovery.

It is interesting to note that these two techniques, which take opposite starting points and use dissimilar methods, achieve closely comparable results on varied inputs in the range we believe most interesting. The top-down approach, which in this range seems slightly inferior to the bottom-up heuristic, offers more promise in applications where the penalty for constructing extra strings is less than we assume here. In the simple cases of regular expressions, where we can bound the optimum solution, both our methods approach these bounds. Our techniques extend the range of combinatorial libraries which can be synthesized by hand by a factor of perhaps 3 to 10. It also produces efficient schedules for larger inputs exceeding 10,000 bases, for which an automated procedure may be necessary to implement the resulting production schedules.

Several interesting open problems remain:

- Whenever a full set of specified molecules cannot be fabricated within a given resource constraint, we seek to synthesize as large a subset as possible. How best can we exploit this freedom?
- The split-synthesis protocol provides the freedom to grow one residue using a subset of more than one amino acid. For example, extending the beads in a chamber using equal amounts of two different reagents will extend half of the molecules by one residue, and half by the other. Since each bead contains a large number of molecules, we get equal numbers of copies per bead. How best can we exploit this freedom?
- Similarly, the protocol provides the freedom to construct longer molecules on beads, each of which contains as submolecules representatives of more than one motif. This significantly reduces the number of beads needed, at the cost of more complex synthesis design. How best can we exploit this freedom?
- Theorem 2 in the Appendix proves the hardness of a restricted form of partitioning into complete bipartite subgraphs. However, the proof does not resolve the most interesting case, namely whether we can cover the graph by $\leq$ 2-stars. It is not clear if this follows by an alternate reduction, or whether a polynomial-time algorithm exists for this problem.

Acknowledgements

We thank Jonathan Montague and Peter Nestler of Cold Spring Harbor Laboratory for useful discussions concerning split-synthesis and combinatorial chemistry.

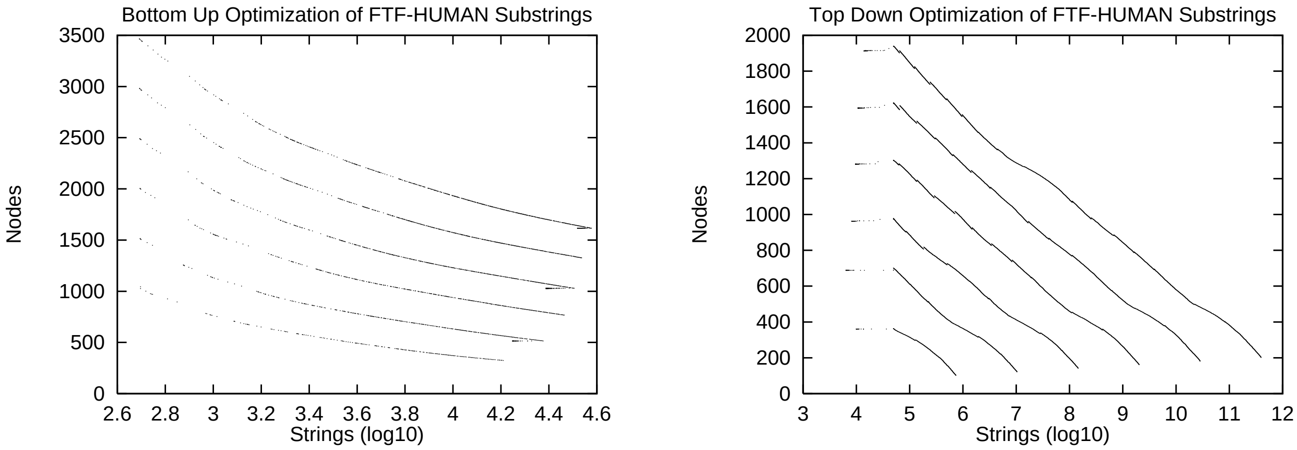

Figure 9: Bottom up and top down synthesis of all k-peptides of FTF-HUMAN, $k = 10$, 9, 8, 7, 6, and 5.

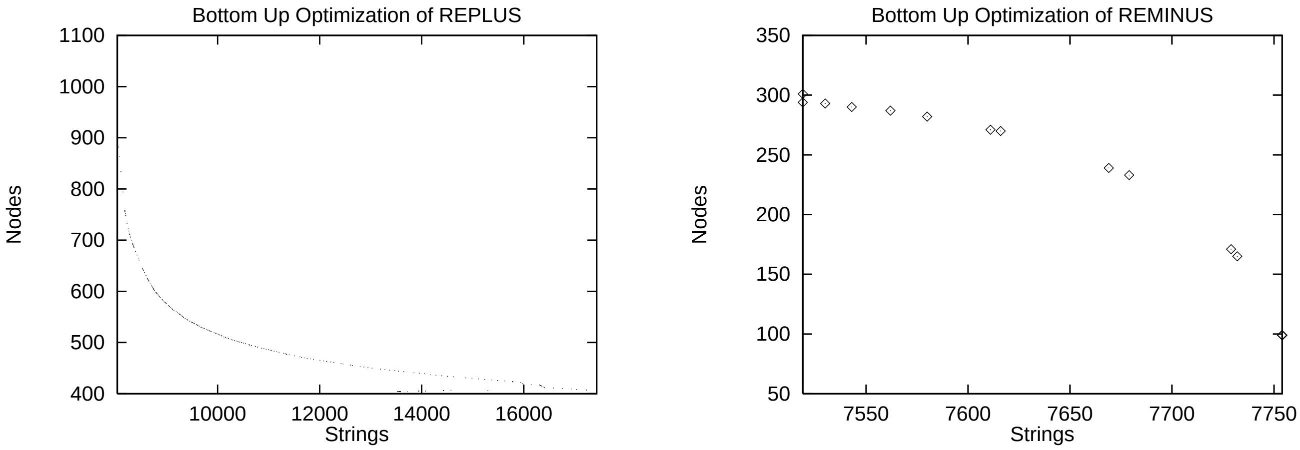

Figure 10: Synthesis of the language of 7,776 strings defined by $((a + b + c)(a + b))^5$ with 256 randomly generated strings added (left) and 256 randomly selected strings deleted (right).

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, Englewood Cliffs NJ, 1993.

[2] R. Bradley and S. Skiena. Fabricating arrays of strings. In *Proc. First Int. Conf. Computational Molecular Biology (RECOMB '97)*, pages 57–66, 1997.

[3] P. Erdos and L. Pyber. Covering a graph by complete bipartite graphs. *Discrete Mathematics*, 170:249–251, 1995.

[4] S. Fodor, J. Read, M. Pirrung, L. Stryer, A. Lu, and D. Solas. Light-directed, spatially addressable parallel chemical synthesis. *Science*, 251:767–773, 1991.

[5] A. Furka, F. Sebestyen, M. Asgedom, and G. Dibo. General method for rapid synthesis of multicomponent peptide mixtures. *Int. J. Peptide Protein Research*, 37:487–493, 1991.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP- completeness.* W. H. Freeman, San Francisco, 1979.

[7] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of machines and computations*, pages 189–196. Academic Press, New York, 1971.

[8] E. Kousofios and S. North. Drawing graphs with dot – dot user's manual. Technical Report, AT&T Bell Laboratories, 1993.

[9] K. Lam, S. Salmon, E. Hersh, V. Hruby, W. Kazmierski, and R. Knapp. A new type of synthetic peptide library for identifying ligand-binding activity. *Nature*, 354:82–86, 1991.

[10] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, Fort Worth TX, 1976.

[11] M. Lebl, V. Krchnak, N. Sepetov, B. Seligmann, P. Strop, S. Felder, and K. Lam. One-bead-one-structure combinatorial libraries. *Biopolymers (Peptide Science)*, 37:177–198, 1995.

[12] L. Lovasz. Communications complexity: A survey. In *Paths, Flows, and VLSI Layout*, pages 235–265, 1990.

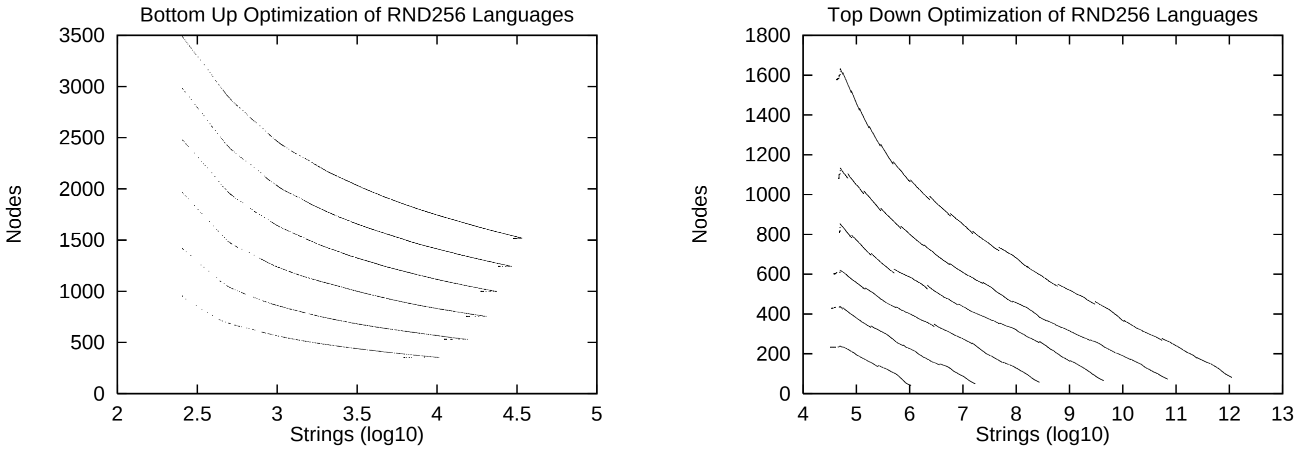

Figure 11: Bottom up and top down synthesis of 256 random k-strings over $\{A, C, G, T\}$, $k = 20$, 18, 16, 14, 12 and 10.

[13] D. Margaritis and S. Skiena. Reconstructing strings from substrings in rounds. In *Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS)*, pages 613–620, 1995.

[14] L. Pyber. Covering the edges of a graph by ... In *Colloquia Mathematica Societatis Janos Bolyai 60; Sets, Graphs and Numbers*, pages 583–610, 1991.

Appendix: Hardness of Partitioning into 3-Stars

The problem of partitioning the edges of a graph into a minimum number of complete bipartite graphs is known to be NP-complete [6]. As an approach towards dealing with this problem in practice, here we consider the complexity of the problem of partitioning graphs into a minimum number of restricted complete bipartite graphs.

For sufficiently restricted classes of graphs, finding such a partitioning is indeed tractable. A *1-star* is a complete bipartite graph of the form $K_{1,x}$ for any x. The problem of partitioning a bipartite graph into the minimum number of 1-stars is identical to finding the minimum vertex cover of a bipartite graph, which is identical to the size of the maximum matching in the graph and can be efficiently computed [10, 1].

In general, a *k-star* is a complete bipartite graph of the form $K_{k,x}$ for any x. Here we consider the problem of partitioning the edges of a bipartite graph into the minimum number of complete bipartite graphs whose smallest part is at most k.

Theorem 2 *The problem of partitioning the edges of a bipartite graph into the minimum number of complete bipartite graphs whose smallest part is at most* 3 *is NP-complete.*

Proof: Clearly the problem is in NP. To show hardness, we show that 3-SAT $\propto$ minimum 3-star partition.

For each boolean variable x_i in the given 3-SAT instance, we construct an instance of the six-vertex, eight-edge gadget in Figure 13. This gadget can be partitioned into two or fewer 3-stars in only two ways, both comprising a $K_{2,3}$ and a $K_{1,2}$. We let one of the degree-two vertices denote the literal x_i, while the other denotes the literal $\bar{x}_i$.

Each clause $C_j = (v_1, v_2, v_3)$ will be represented by a subgraph $K_{4,4}$, where three of the vertices on the left side of the partition will be the appropriately labeled vertices of the variable gadgets. The other five vertices will be unique to the clause gadget C_j, so the final graph contains a total of $5|C| + 6|V|$ vertices. Note that each clause gadget requires at least two $\leq$ 3-stars to cover, unless all the edges on at least one of the variable vertices is otherwise covered, leaving a single $\leq$ 3-star. The graph associated with the 3-SAT instance $\{\{X_1, X_2, X_3\}, \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}\}$ appears in Figure 13.

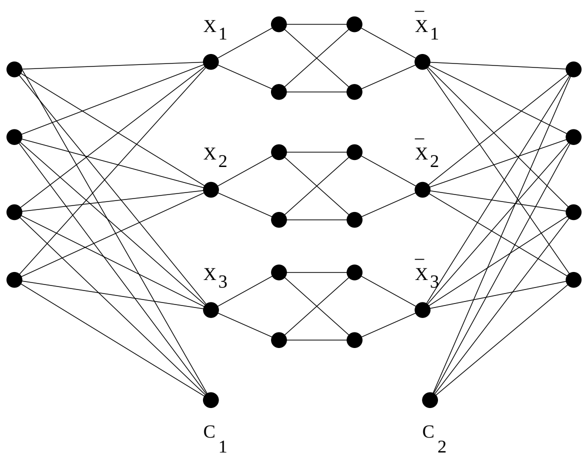

Figure 13: Bipartite graph representing the 3-SAT instance $\{\{X_1, X_2, X_3\}, \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}\}$.

We claim that the resulting graph can be partitioned into $2|V|+|C| \leq$ 3-stars iff the set of clauses it is constructed from are satisfiable. If the clauses are satisfiable, an appropriate bipartite clique partition can be constructed by placing 1-stars on all true literals' vertices. Since every clause will have at least one of its vertices' edges covered, the remaining clause gadget can be covered using one $\leq$ 3-star. The remaining pieces of the variable gadgets can each be covered by $K_{2,3}$.

Now suppose that we partitioned the edges into $2|V|+|C|$ $\leq$ 3-stars. Since we need at least two stars for each variable gadget and at least one star for each clique gadget, there is no slack for any partitioning which does not place a 1-star over one literal vertex of each variable gadget. Let this define the truth assignment. It must be satisfying to remove at least one vertex's edges from each clause gadget, leaving the rest to be covered by a 1-, 2-, or 3-star. ∎