Designing a Multiroute Synthesis Scheme in Combinatorial Chemistry

Adi Akavia,[†] Hanoch Senderowitz,[‡] Alon Lerner,[§] and Ron Shamir*[,§]

School of Computer Science, Tel Aviv University, 79978, Israel, Computer Science and Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel, and Predix Pharmaceuticals Ltd., S.A.P Building, 3 Hayetzira Street, Ramat Gan 52521, Israel

Received September 19, 2003

Solid-phase mix-and-split combinatorial synthesis is often used to produce large arrays of compounds to be tested during the various stages of the drug development process. This method can be represented by a synthesis graph in which nodes correspond to grow operations and arcs to beads transferred among the different reaction vessels. In this work, we address the problem of designing such a graph which maximizes the number of produced target compounds (namely, compounds out of an input library of desired molecules), given constraints on the number of beads used for library synthesis and on the number of reaction vessels available for concurrent grow steps. We present a heuristic based on a discrete search for solving this problem, test our solution on several data sets, explore its behavior, and show that it achieves good performance.

1. Introduction

Drug development is a long and expensive process; hence, methods with the potential of accelerating it are of the utmost importance. Combinatorial chemistry[3,5,9,11,13,15−17,22,24,26,29] can greatly accelerate the lead discovery phase by producing large arrays of compounds that could be screened for biological activity in a high-throughput screening (HTS) manner. Unfortunately, the very nature of traditional solid-phase mix-and-split synthesis (see Figure 1b), which produces highly similar compounds, contrasts with the requirements of the lead discovery phase, in which highly diverse sets of compounds are usually desired. Such diversity requirements could be met through parallel synthesis (see Figure 1a), but this strategy is limited in the number of compounds it can produce.

In this work, we take the first steps toward bridging the seemingly opposing characteristics of parallel and combinatorial synthesis by presenting a method for designing mix-and-split-based synthesis schemes that maximize the number of produced target compounds, namely, compounds from an input library of desired molecules. This method could be combined with a diversity selection algorithm[1,4,14,18,19,23,25] to produce a unified scheme that facilitates synthesis of large and diverse compound libraries.

Our work is based on the multiroute synthesis scheme originally proposed by Cohen and Skiena.[6] This scheme is a generalization of both the mix-and-split and the parallel synthesis methods, and similar to mix-and-split synthesis, it is a process by which a large set of compounds is synthesized in a combination of mix, split, and grow steps. However, the mix steps here need not be a mixing of all the previously produced compounds, but rather of any desired combination of the previous subsets.

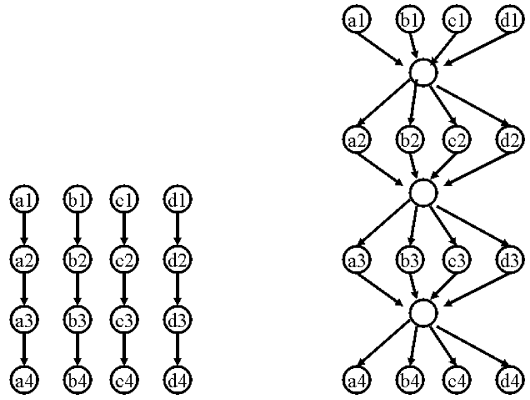

Figure 1. (a) Parallel Synthesis: The process described by this graph produces four strings, $a_1a_2a_3a_4$, $b_1b_2b_3b_4$, $c_1c_2c_3c_4$, and $d_1d_2d_3d_4$. Building units (node labels) are added according to the order in each chain. (b) Mix-and-split synthesis: the empty nodes are mix steps. The labeled nodes are grow steps. Split steps are denoted by arrows emanating from a mix node. The process described by this graph produces 256 strings.

A description of a multiroute synthesis may be given in a synthesis graph, as presented in the Methodology Section. A synthesis graph (see Figure 2a) is a labeled, directed, acyclic graph composed of layers, each describing the grow operations for a position in the target compounds. The nodes in the graph correspond to the grow operations, and their labels indicate the appended unit. The arcs of the graph correspond to beads transferred among the different reaction vessels; that is, if node v has incoming arcs from nodes u_1, ..., u_k, then the mixing step takes compounds from nodes u_1, ..., u_k into node v. Combined together, the mixing step takes compounds from all nodes with incoming arcs to v, and the grow step appends to each of these compounds a unit given by the label of v.

The multiroute synthesis is appealing, because it enables synthesis of libraries that are far larger than those producible

* To whom correspondence should be addressed. E-mail: rshamir@tau.ac.il.

[†] The Weizman Institute.

[‡] Predix Pharmaceuticals Ltd..

[§] Tel Aviv University.

10.1021/cc034045i CCC: $27.50
Published on Web 05/18/2004

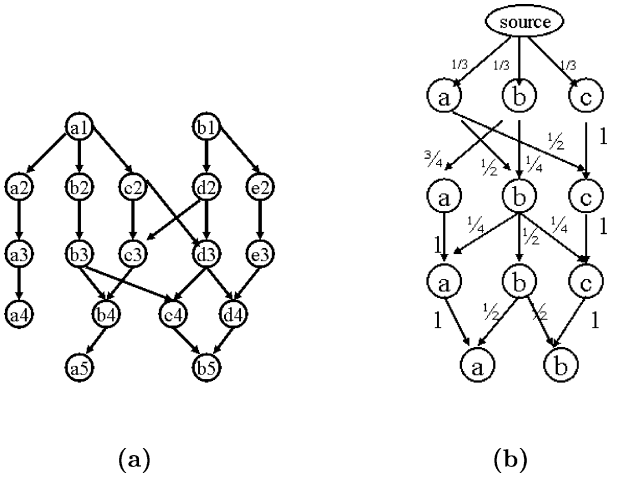

Figure 2. (a) Multiroute synthesis: The produced strings are $a_1a_2a_3a_4$, $a_1b_2b_3b_4a_5$, $a_1b_2b_3c_4b_5$, $a_1c_2c_3b_4a_5$, $a_1c_2d_3c_4b_5$, $a_1c_2d_3d_4b_5$, $b_1d_2d_3c_4b_5$, $b_1d_2c_3b_4a_5$, $b_1d_2d_3d_4b_5$, and $b_1e_2e_3d_4b_5$. (b) Weighted multiroute synthesis: the weight of an arc (u, v) represents the fraction of beads from u transferred to v.

by parallel synthesis and, at the same time, far more diverse than those producible by mix-and-split synthesis. However, in contrast to parallel or mix-and-split synthesis, for a fixed assignment of labels to nodes, many multiroute synthesis graphs could be designed, each leading to a different set of output compounds. Thus, in order to locate the synthesis graph that produces the largest number of target compounds from a given input set, a search should be performed.

In their work, Cohen and Skiena[6] considered only unweighted synthesis graphs, whereas we also consider the weighted case. In weighted multiroute synthesis, a nonuniform distribution of the beads transferred from a node (i.e., reaction vessel) to its descendants is allowed. Weighted multiroute synthesis is modeled by a weighted synthesis graph. A weighted synthesis graph is a synthesis graph with nonnegative arcs weights (see Figure 2b) in which the weight of an arc (u,v) indicates the number of beads that are transferred from node u to v.

In many real-life situations, the laboratory constraints on the number of available reaction vessels (corresponding to the number of nodes in any single layer) and the number of beads used are quite rigid. On the other hand, the target set of strings is often heuristically designed, so producing all of the strings may not be critical. Moreover, avoiding some target strings can greatly reduce the required number of beads and reaction vessels. Hence, we explore the max string synthesis problem, in which the goal is to produce as many target strings as possible within the above constraints (see the Methodology Section). We present a heuristic based on a discrete search for solving this problem, test the heuristic on several data sets, explore its behavior, and show that it achieves good performance. Test runs were performed on sets of peptides since these form natural candidates for all synthesis methods (i.e., parallel, mix-and-split, multiroute) discussed in this work.

In addition to Cohen and Skiena's multiroute synthesis scheme, several other strategies were considered for designing mix-and-split libraries.[8,21,27] Most attempted to factor into the design strategies the cost of mixture deconvolution. In contrast, the present work does not consider deconvolution issues and assumes that either (a) the beads are tagged or (b) the beads are large enough and so contain enough material to allow for a direct product identification via analytical methods. A second major difference between our approach and those reported in the literature is that while previous strategies aimed at finding the optimal mix-and-split scheme that would produce the entire target library, we recognize the fact that giving up on some of the target sequences may greatly simplify the mix-and-split scheme. Thus, we look to maximize the number of target strings produced, subject to synthesis constraints, rather than finding the best solution for the mix-and-split scheme that would generate all target strings.

This paper is organized as follows: In the Methodology Section, we describe a model of the multiroute synthesis process; present the problem that we study, max string synthesis; and describe a heuristic for solving it. In the Implementation Section, we give some details on our implementation, and in Results and Discussion, we present results from extensive experiments we performed in order to evaluate its performance and time requirements.

2. Methodology

In this section, we present our methodology for solving the max strings synthesis problem. We begin by describing the weighted synthesis graph, by which we represent the synthesis process; proceed by giving a mathematical formulation of the max string synthesis problem; and then present the algorithm we developed for solving it.

Synthesis Graph Model. We model the synthesis process by a weighted graph (for an introduction to graphs in computer science, see ref 7). A weighted synthesis graph (see Figure 2b) is a layered graph, $G = (V, E)$, with a weight function that assigns a positive weight to each arc, and a source node, which is connected to all nodes in the first layer. All nodes except the source node represent grow steps and are labeled by the appended unit. The arcs are directed from one layer to the next, and they represent transfer of beads between grow operations. The weight of an arc (u,v) represents the fraction of beads from node u that are transferred to v.

Let $P(G)$ denote the set of all paths in G starting at the source and ending with a node in the last layer. Each path p corresponds to the string σ obtained by concatenating the labels along its nodes. The weight of a path, p, denoted by weight(p), is the product of its normalized arcs' weights. The weight of a string, σ, denoted by weight(σ), is the sum of weights of all paths in $P(G)$ corresponding to σ. Note that weight(σ) is equal to the fraction of beads expected to hold σ at the end of the synthesis process described by G. Namely, when b beads are used, weight(σ) $\times$ b beads are expected to hold σ. Hence, we say that σ is produced by G when using b beads if weight(σ) $\times$ $b \geq 1$. The set of all strings σ produced by G when using b beads is called the language produced by the graph, and is denoted by $L(G, b)$.

We note that a synthesis graph represents a stochastic process, because each split step randomly divides the set of

beads into subsets (albeit according to the arcs weights); therefore, $L(G, b)$ corresponds to the expected set of strings produced by G when using b beads, not necessarily to the strings that are actually produced. When applying the synthesis process, one can ensure that all the expected strings are produced with high probability by taking redundant beads (i.e., by using $b' > b$ beads to increase the chance that the entire set $L(G, b)$ is produced). The amount of redundancy needed is explored in ref 28.

Relevant Parameters. Let S be a set of target strings (a library). Following Cohen and Skiena,[6] we focus on the problem of finding a synthesis scheme for S; however, when considering the real-world version of the problem, there are several different possible formulations to it. These formulations depend on the choice of parameters to be constrained and those to be optimized. Generally speaking, the relevant parameters we focus on are

1. $|V|$: the number of nodes in the graph (which corresponds to the number of grow steps). This number can be estimated by the width of the graph (which corresponds to the number of parallel grow steps) and the depth of the graph, which is determined by the length of the strings in S.

2. b: the number of beads used in the synthesis process.

3. $|P(G)|$: the number of paths in the graph (which roughly corresponds to the number of needed beads in the synthesis process).

4. $|L(G,b)\cap S|$: the number of target strings that are produced by the graph.

In the above list of parameters, the first three are to be minimized, while the fourth should be maximized. We have, therefore, four parameters, and in any optimization problem, some may be bounded (or set to a fixed value, or penalized) and some optimized. These variants give different problems, which might vary greatly in complexity of their solutions. In the following, we briefly consider examples of such variants.

First, let us consider two such variants, which yield easy-to-solve problems. If we must produce all target strings, while there is no penalty on the number of nodes and the number of paths is to be minimized, the solution is immediate: use parallel synthesis to produce the target strings. If producing all target strings is required, there is no penalty on the number of paths, and the number of nodes is to be minimized, the solution is again obvious: mix-and-split synthesis.

Second, we consider hard-to-solve variants. One variant which has been shown to be NP-hard by Cohen and Skiena[6] is the problem in which producing all target strings is required, while the number of paths is bounded, and the goal is to minimize the number of nodes. In the rest of the paper we focus on a different variant of the problem: the max strings synthesis.

Max Strings Synthesis. Max strings synthesis maximizes the number of produced target strings while constraining the number of nodes in each layer and while limiting the number of beads. Namely, given a set S of target strings and two positive integers w and b, the goal is to find a weighted synthesis graph G of width, at most, w, that maximizes $|L(G,b)\cap S|$ (see example in Figure 3).

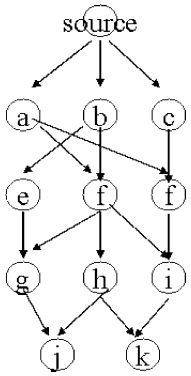

Figure 3. An example of max strings synthesis problem and solution. Consider an instance of the max strings synthesis problem with constraints $w = 3$ and $b = 11$, and with a target library $S = \{afhj, afik, begj, bfgj, bfhk, cfik, cfgj\}$. The above graph is an example of a solution to this problem. It has width 3, and when using 11 beads, it produces all strings in S except *cfgj*. In addition, it produces the nontarget strings *afgj*, *afhk*, *bfhj*, and *bfik*.

Our motivation for defining the max strings synthesis problem is the observation that in many real-life situations, the laboratory constraints on the number of reaction vessels (corresponding to the number of nodes in each layer) and the number of beads used are quite rigid. On the other hand, the target set of strings is often heuristically designed, so producing all of them may not be critical. Often, avoiding some target strings can reduce the number of paths and the number of nodes sharply. Hence, we define the max strings synthesis problem, in which the goal is to produce as many target strings as possible subject to the above constraints. Max strings synthesis was proven to be NP-hard in ref 2.

Algorithm. In this section, we present a heuristic for solving max strings synthesis that operates by a discrete search over the space of synthesis graphs. Searching for a good synthesis graph, we face two opposing goals: naturally, a good synthesis graph should have as many paths corresponding to target strings as possible, while at the same time, since the number of beads is bounded, it should have as few paths for nontarget strings as possible. To balance these two opposing goals, we impose a reduction of the number of paths by limiting the search to graphs which are composed of disconnected layered subgraphs, called slices, each containing at most one copy of each label in each layer. This limits the number of paths, since we allow no arcs connecting nodes in different slices (see Figure 4).

Our algorithm for solving max strings synthesis is composed of two main procedures, which are alternately applied: a slice initialization procedure and an optimization procedure. It maintains a set, S', of target strings that are not produced by the graph built thus far. Initially, $S' = S$. In the slice initialization procedure, node labels and arc weights are determined for a slice of the graph. The choice of labels and weights is done in accordance with sequence frequencies in the set S'. In the optimization procedure, we aim at maximizing the number of target strings that are produced by all slices initialized thus far. The optimization procedure operates in two main steps: (1) arcs are deleted from the graph until the number of paths is below the bound b on the number of beads, and (2) once there are no more than b paths in the graph, arcs weights are recalculated so that the new

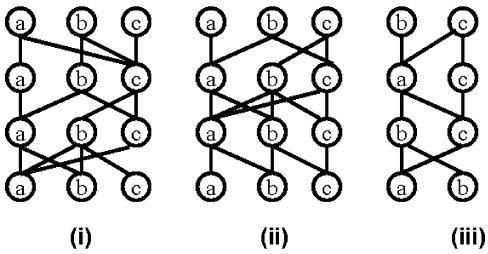

Figure 4. A four-layer graph of width 8 and alphabet of three letters, a, b, and c. The graph consists of three slices, (i), (ii), and (iii), with at most one copy of each label in each layer. Due to the constraint imposed on the width of the graph, the third slice consists of two nodes only, labeled with letters that are the most frequently occurring at the corresponding sequence positions in the target strings not generated by slices (i) and (ii). See the Slice Initialization section for more details.

weights allow all the strings corresponding to the paths in $P(G)$ to be produced. The slice initialization and the optimization procedures are alternately applied until all slices are initialized and optimized.

Slice Initialization. Let S' denote those strings in S that are not produced by the current graph. In each layer, l, of the slice, nodes are labeled by the letters in position l along the strings composing S' (note that the slice width in layer l does not exceed the number of letters in position l in S'). When the number of nodes is equal to the number of letters, each letter is given as a label to a distinct node. Otherwise, some letters are not assigned as labels (since the number of nodes is less than the number of letters). Thus, assigning node labels already determines that some strings in S' will not be produced by this slice. To minimize the number of such strings, we assign as labels the most frequent letters in the set obtained from S' after discarding all strings that do not have appropriate letters in previously assigned layers. For example, assume all strings in S' that have the letter "*b*" in the second sequence position also have the letter "*a*" in the first sequence position. In this case, if *a* was not given as a label to any node in the first layer (due to its low frequency), then *b* is not given as a label to any node in the second layer (although *b* might be very frequent in the second layer of S').

Once all node labels are assigned, let S'' denote the set S' after discarding all strings that cannot be produced by the slice (because they lack appropriate labels). Determining the arcs weights is done as follows: We consider every two nodes u, v in consecutive layers of the slice. Let label(u) and label(v) denote the labels of u and v, respectively. To determine the weight of the arc (u, v), we consider the strings in S'' with letters label(u), label(v) in the positions corresponding to the layers of u and v, respectively. The assigned weight is the fraction of these strings out of all strings in S'' with the letter label(u) in the position corresponding to the layer of u. Note that if there are no such strings, then the weight of the arc is 0, which is regarded as a nonexistent arc. This form of initialization was the best of all the alternatives we considered (see ref 2).

Optimization. Optimizing the number of target strings produced by the current graph (namely, all slices initialized thus far) is done in two main steps: arc deletion, and arc weights recalculation.

Arc Deletion. In this step, arcs are deleted from the graph until the number of paths falls below the constraint b on the number of beads. When choosing arcs to be deleted, we would like to maintain as many paths corresponding to target strings as possible while reducing the total number of paths. For this purpose, we assign a score to each arc and delete the arc with the worst score. We examined three scoring functions: prob score, path score, and lookahead score.

To simplify the description of the scoring functions, we define two terms: a target path is a path $p \in P(G)$ corresponding to some target string, and a superfluous path is a path $p \in P(G)$ that does not correspond to any target string. The scoring functions we examined are defined as follows.

- **Prob Score.** The prob score of arc e is the total probability of the target paths using it, namely,

$$\text{prob score}(e) = \sum_{p \in \text{ TargetPaths}(e)} \text{weight}(p)$$

where TargetPaths(e) denotes the set of target paths passing through the arc e. Note that the above summation can be calculated efficiently using the fact that each slice may contain at most one copy of each target path: by scanning a target path within a slice, one can readily find out if that path appears, and if so, increase the weights of all the arcs used in this path. The time required for scanning all target paths against all slices is proportional to the input size times the width, w.

- **Path Score.** The path score of arc e is the ratio between the number of target paths and the number of superfluous paths passing through it, namely,

$$\text{path score}(e) = \frac{\text{no. of target paths passing through } e}{\text{no. of superfluous paths passing through } e}$$

Computing the numerator can be done in the same fashion as for the prob score. Although there may be an exponential number of superfluous paths, their number can be efficiently computed using the observation that the number of superfluous paths through an arc equals the total number of paths passing through it minus the number of target paths passing through it. The total number of paths passing through an arc (u, v) is the product of the number of its incoming and outgoing paths. These values can be recursively computed: Denote by out(u, v), the number of outgoing paths starting with the arc (u, v). Then, out(u, v) = 1 for arcs (u, v) ending in the last layer (all compounds from the last layer are outgoing to the pool of all produced compounds); otherwise, out(u, v), is the sum of out(v, w) over all arcs (v, w) emerging from v. The number of incoming paths is similarly computed.

- **Lookahead Score.** Consider a string σ that corresponds only to a single path $p \in P(G)$. When deleting an arc e on p, the string σ can no longer be produced by the graph. Therefore, any other arc that was used only in paths corresponding to σ may also be deleted. In the lookahead score, we take into account the total effect of the deletion of the arc, not only the counts of target and superfluous paths passing through it. Thus, the lookahead score of an arc is the ratio between target and superfluous paths eliminated from the graph by the deletion of the arc.

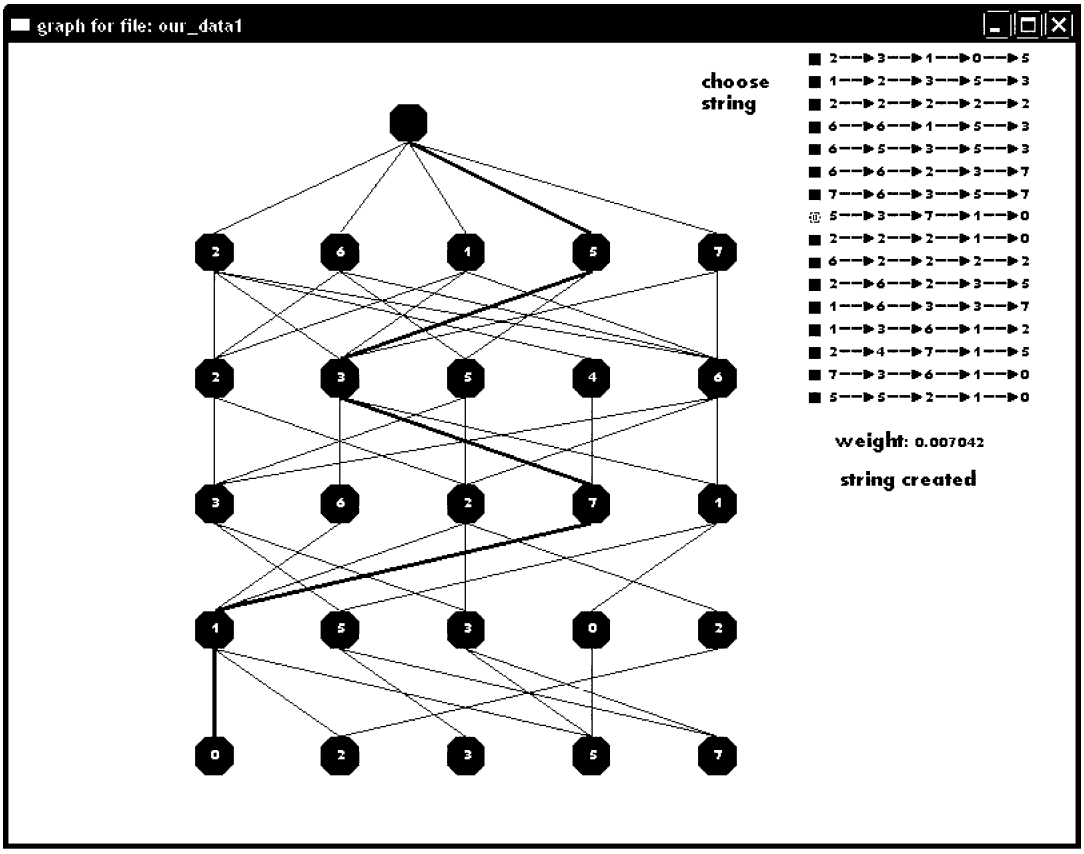

Figure 5. Graphical representation of a synthesis graph and the strings produced by it, as produced by our program. Appended units (node labels) are given by numbers. Target strings that correspond to a path in the graph appear on the right-hand side. When a path corresponding to one of the target strings is chosen (marked by a lighter square to its left), its path is drawn in bold, and the weight of the path appears below the list. The indication "string created" means that given the bound on the number of beads, this string is actually produced by the graph.

$$\text{lookahead score}(e) = \frac{\text{no. of target paths eliminated as a result of deleting } e}{\text{no. of superfluous paths eliminated as a result of deleting } e}$$

Recalculating Arcs Weights. Once the number of paths in the graph is no more than the number of beads, the arcs weights are reassigned to be

$$\text{weight}(v, u) = \frac{\text{outgoing}(u)}{\text{outgoing}(v)}$$

where outgoing(v) is the number of paths outgoing from node v (which can be efficiently calculated similarly to the calculations of the Path Score above). These arc weights guarantee that the weight of each path in $P(G)$ is at least $1/b$, and thus, all the paths in the graph are produced by it.[2]

In summary, the high level description of the algorithm is as follows:

While there is another slice to initialize

a. Initialize next slice: assign node labels and arc weights.

b. Optimize current graph (namely, all slices initialized thus far):

i. While number of paths exceeds number of beads: compute arc scores and delete an arc with the worst score.

ii. Recalculate arc weights.

3. Implementation

So far, we have described our algorithms and the synthesis graph under the assumption that all target strings are of the same length. This was done in order to simplify the description. In practice, we also handle string sets with varying lengths. For this purpose, we add to the synthesis graph auxiliary nodes, called phantom nodes, with which we reduce the problem to the simplified case of all strings being of the same length. For details we refer the reader to ref 2.

Our algorithm is implemented using the C++ language. As a means of visualizing the resulting synthesis graph, we implemented a graphical interface (Figures 5, 6). This graphical interface presents the output graph (labels and arcs weights), together with some additional information, such as the list of strings produced by the graph, their weights, and the paths through which they were produced.

4. Results and Discussion

To test both the performance and the time requirements of our algorithm for solving the max strings synthesis problem, we performed extensive experiments on different types of data sets. In this section, we describe the data sets we used and compare the results of running the different versions of our algorithm. For the version that proved best (the lookahead score), we then present more extensive results.

Data Sets. We worked with a parent library, which contained all possible sequences of length 5 generated by the 10 natural amino acids (in parentheses, single letter code): alanine (A), arginine (R), asparagine (N), aspartic acid (D), cysteine (C), glutamine (Q), glutamic acid (E),

glycine (G), histidine (H), and isoleucine (I). Each sequence was characterized by a set of 30 descriptors, and following principal component analysis (PCA), it was found that six principal components covered more than 90% of the variance in the original data set. Sets of sequences were diversity-selected from the space defined by the above six PCs by the MaxMin function using 100 000 Monte Carlo steps with 10 000 idle steps as a termination criterion. All these calculations were performed with Cerius^2 version 4.5.[10]

The MaxMin function belongs to a class of algorithms[1,4,14,18,19,23,25] that aim to select from within a parent library subsets of compounds that are as different from one another as possible. Such diverse subsets are expected to display a diverse spectrum of biological activities and, consequently, form good candidates for biological screening as part of lead discovery projects.

We note that diversity-selected data sets are less likely to be produced by mix-and-split related combinatorial chemistry because, on average, they have significant sequence dissimilarities. In this sense, such data sets present a greater challenge (compared with focused or random libraries) for any algorithmic solution to the synthesis design problem. Still, we chose to handle such data sets in order to comply with common practice in the early stages of the drug design process.

Using this diversity selection method, we chose 50 small data sets, each containing 96 compounds. We also chose 20 large data sets, each containing 1000 compounds. In addition, we designed synthetic data sets, each based on a set of 60 compounds from the parent library, of which 55 compounds could be produced by a synthesis graph with the basic parameters of 10 000 beads and width 10. We call these 60 compounds "structured compounds". The 60 compounds were chosen such that a minimum synthesis graph of width 10 producing all of them has out-degree 6 in each node leading to 10×6^4 paths, which is more than the number of beads (10 000). By eliminating five paths, we could obtain a synthesis graph producing 55 compounds, in which the out-degree of half of the nodes is reduced to 5. This graph has only $5 \times 5^4 + 5 \times 6^4$ paths (which is less than the number of beads). This design gave us a lower bound on the number of paths in the optimal solution and, hence, a way to evaluate the performances of our algorithms in an absolute manner. On top of theses 60 compounds, we added different amounts of "noise", that is, extra compounds, which were randomly chosen from the parent library.

In summary, this is a list of the data sets we used:

(i) 50 sets of 96 compounds, diversity-selected from the parent library (termed "96 real data"),

(ii) 20 sets of 1000 compounds, diversity-selected from the parent library (termed "1000 real data"),

(iii) 10 sets of 60 structured compounds plus 10 "noise compounds" (termed "70 synth data"),

(iv) 10 sets of 60 structured compounds plus 20 "noise compounds" (termed "80 synth data"),

(v) 10 sets of 60 structured compounds plus 30 "noise compounds" (termed "90 synth data"), and

(vi) 10 sets of 60 structured compounds plus 40 "noise compounds"(termed "100 synth data").

Table 1. Statistics on the Performance of All Algorithms on the Different Types of Data Sets[a]

	prob	path	lookahead
1000 real data	340.8 (13.18)	357.8 (11.61)	357.8 (11.32)
96 real data	86.0 (1.68)	85.0 (2.12)	87.0 (1.57)
100 synth data	80.6 (1.64)	78.4 (1.50)	81.5 (1.43)
90 synth data	75.0 (1.24)	73.1 (1.79)	75.7 (1.15)
80 synth data	68.3 (1.33)	67.2 (1.31)	68.7 (0.94)
70 synth data	61.6 (0.84)	61.2 (0.91)	61.9 (0.87)

[a] Each line gives the average and standard deviation (in parentheses) of the number of target strings produced by our algorithm when run with the different scoring functions.

Results and Discussion. Statistics for the performance of the algorithm (when using each of the three scoring functions) on the different data sets are given in Table 1. For each type of data set, we present the average and standard deviation of the number of target strings that were produced by the algorithm. We ran the algorithm with the following basic parameters: number of beads 10 000 and width 10.

The results on the synthetic data were encouraging. For each of the data sets, the number of generated strings is >55, the number of strings we knew to be producible. In fact, in all cases, the number was larger, because additional strings from the "noise" were generated. For the 96 and 1000 real data sets, it is harder to estimate the performance of our algorithm, because the maximum number of target strings that could be produced is not known. We see that on average, more than one-third of the strings in the large data sets (1000 strings) and almost 90% of the strings in the small data sets (96 strings) are produced.

The running time of the algorithm is very fast. On a Pentium III (500 MHz, 0.5GB RAM), when using the lookahead score, the large data sets (1000 strings) require, on average, 45.15 s, while the small data sets (96 strings) require only 0.0032 s.

The lookahead score is slightly better than the other scores. This is not surprising, since this score takes into account the effect of an arc deletion to a deeper horizon. We now examine more thoroughly the behavior of our algorithm when run with the lookahead score and when varying the constraints imposed on the synthesis graph.

First, we explored the difference between the results obtained for the two real data sets, as measured by the coverage, i.e., the percentage of target strings that are actually produced. The difference in the average coverage, 87% for the small data sets (96 strings) and 36% for the large data sets (1000 strings), is easily understood when recalling that the graph parameters (width 10, and number of beads 10 000) were kept the same for both data sets, despite the great difference in their sizes. Figure 7 shows that, indeed, when the width of the graph is increased by a factor of 10, the coverage over the large data sets (1000 strings) becomes close to 80%, as expected. (In addition, when the number of beads is increased 10-fold, the coverage trivially becomes 100%, as in a graph of width 10 and length 5, the full combinatorial library requires no more than 100 000 beads.)

Next, we explored the behavior of our algorithm when different values of the graph width and the number of beads

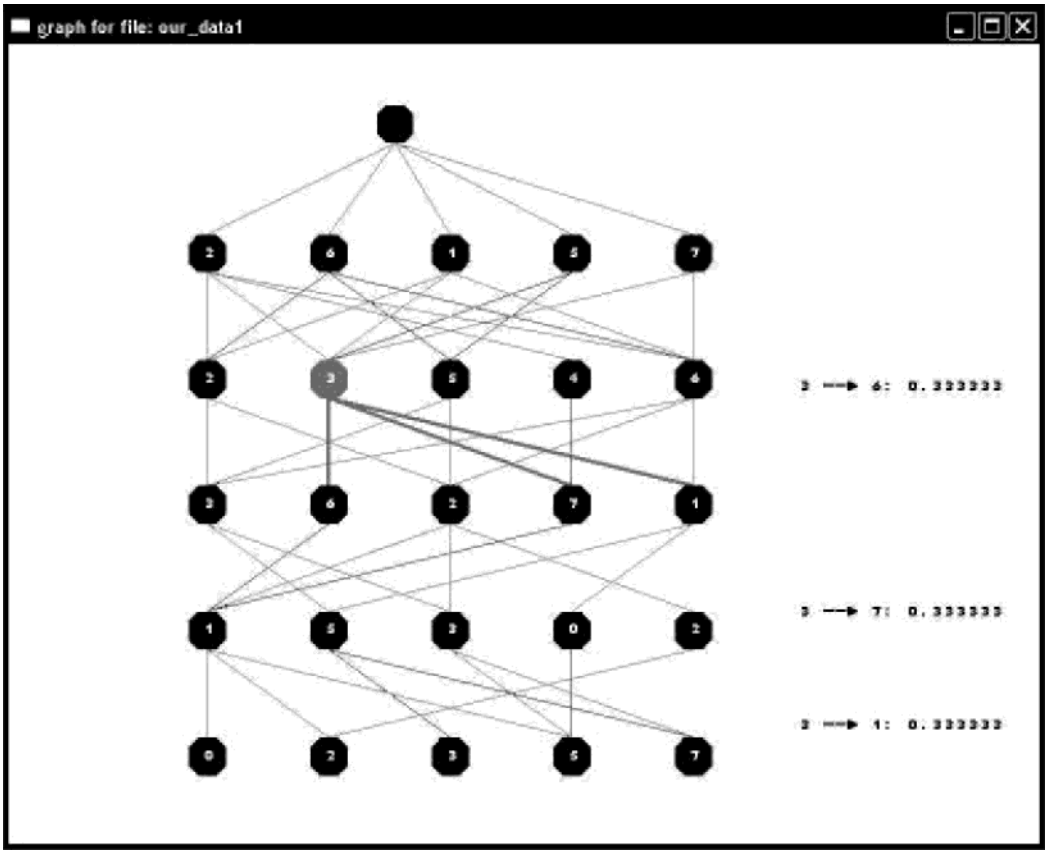

Figure 6. Graphical representation of a synthesis graph and its arc weights, as produced by our program. When a node is chosen (node 3 in layer 2, here), the weights of the arcs emanating from it are listed on the right.

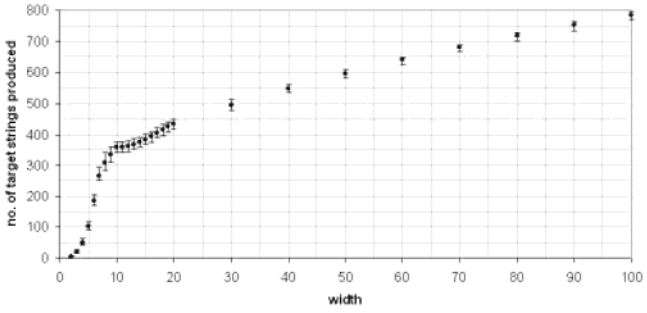

Figure 7. Impact of the graph width on performance. The graph summarizes the results of the lookahead algorithm with different widths on real data sets of 1000 strings. The average (dot) and minimum and maximum (bars) are shown for each width.

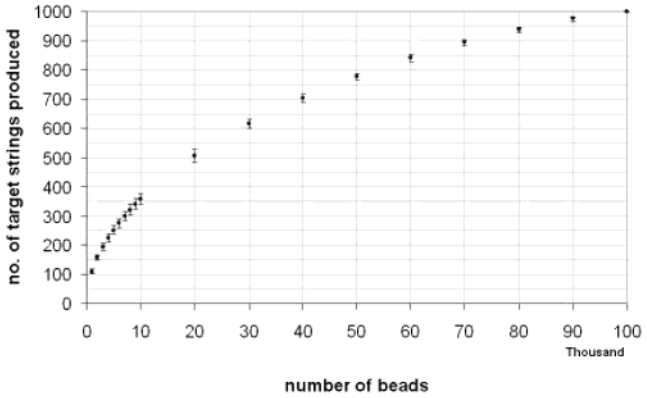

Figure 8. Impact of the number of beads on performance. The graph summarizes the results of the lookahead algorithm with different number of beads on real data sets of 1000 strings. The average (dot) and minimum and maximum (bars) are shown for each number of beads.

were taken. Figure 7 presents the results for the large sets of real data with different widths between 2 and 100. Figure 8 gives the number of produced strings on graphs with width 10 as a function of the number of beads (similar results were obtained for the small data sets and are not shown).

From the above results, it is clear that increasing either the width or the number of beads improves the results. The effect of increasing the number of beads is moderate, whereas increasing the width first causes a rapid exponential-like rise of the number of produced target strings and then a slow rise. The rapid rise continues until $w = 10$, where the alphabet size (the number of distinct amino acids used in the target library) is reached, because each new letter that is added for the first time accommodates many of the target sequences. After all the letters are present, the effect of additional copies of the same letters is more modest. In Figure 9, we explore the tradeoff between those two parameters. Such plots allow the experimentalist to choose the most convenient combination of the two parameters in order to achieve a desired number of target strings.

Because our algorithm is deterministic and greedy, we explored its stability by varying its starting points. For this purpose, from each of the large data sets we generated 950 subsets, each containing 900 randomly selected sequences, and we ran our algorithm on each of those subsets as the target set. Note that each such subset imposes a slightly different starting point, as compared with the one for the full set of 1000 strings. The results obtained from these

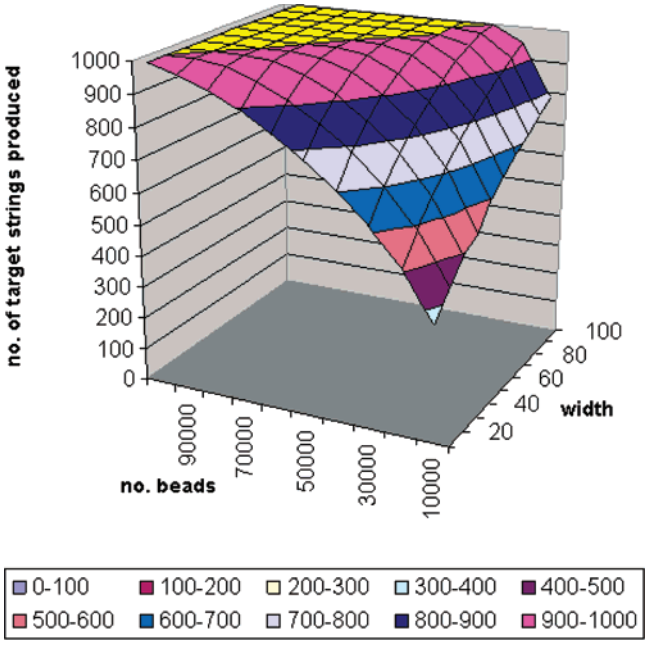

Figure 9. Tradeoff between the width and the number of beads. Average performance of the lookahead algorithm with different widths and different numbers of beads on 1000-string real data sets. Color coding represents the number of produced target strings.

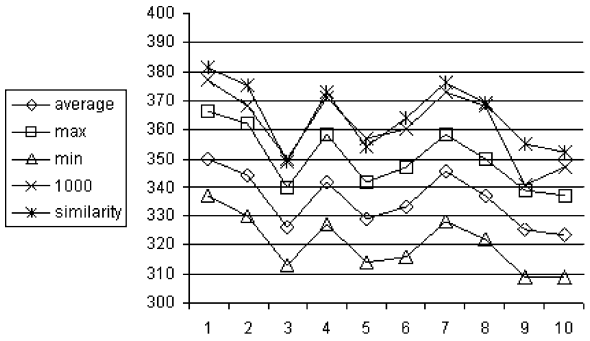

Figure 10. Impact of partial sets. To achieve different starting points, the algorithm was applied to 950 different subsets of 900 target strings, which were chosen out of the 1000-string real data set. This test was performed on 10 sets chosen arbitrarily out of the 20 1000-real-data sets. The *x* axis gives the 10 data sets in arbitrary order. The *y* axis gives the number of produced target strings. The three bottom lines give the average, maximum, and minimum scores obtained. The next line indicates the results on the full set. The top line gives the results on a set of 900 target strings chosen so as to achieve maximum sequence similarity.

random sets have a standard deviation of, at most, 5 strings, which is 0.5% of the full set of 1000 strings, and about 1.5% of the actually produced strings, thus indicating the stability of the algorithm.

In addition, we compared the performance of our algorithm on the random subsets, with its performance on subsets of 900 strings chosen by their high sequence similarity,[2] and found (see Figure 10) that our algorithm always performs better on high-similarity subsets than on the randomly chosen subsets. Interestingly, in 8 out of the 10 data sets we examined, the performance on the high similarity subsets was better than the performance on the full set of 1000 strings. These results give an initial indication of the importance of integrating the diversity-selection methods and the synthesis design methods into a comprehensive scheme for designing libraries that are both diverse and efficiently producible.

An Alternative Approach. One may consider other approaches to solving max strings synthesis. For example, using the same model of the weighted synthesis graph, one can search for the optimum graph using other optimization methods. As a first step toward examining this approach, we devised a continuous optimization algorithm that uses the steepest-descent technique to optimize the arcs weights (while nodes are labeled using the initialization method specified earlier). Our objective in this steepest-descent algorithm was to maximize a function that is a continuous smoothing of the number of target strings produced by the synthesis graph (as determined by the arc weights and node labels).

We tested this algorithm and found that it gives poorer results in comparison with our discrete optimization.[2] However, we observed[2] that a combination of the steepest-descent optimization with an arc deletion heuristic does achieve competitive results, as compared to the discrete algorithm. Further developments of alternative optimization methods for this problem are left to future work.

5. Conclusion

In this work, we presented the max strings synthesis problem, which formulates the question of finding the best multiroute synthesis procedure for a given library of compounds. In solving this problem, we defined the model of the weighted synthesis graph and presented a discrete optimization algorithm searching for the best graph. We tested the discrete algorithm on several types of data sets and on many configurations of the basic constraints and shows that it is stable, behaves as expected (e.g., performs better on sets of similar strings), and achieves good results.

The multiroute synthesis and our method for designing it are quite general and could easily be applied to other synthesis schemes, for example, the string synthesis process,[12] which enables the easy identification of each of the produced compounds (similar to the parallel synthesis), while producing combinatorial libraries (as in mix-and-split or multiroute synthesis).

Acknowledgment. R.S. was supported in part by the Israeli Science Foundation (Grant no. 309/02).

References and Notes

(1) Agrafiotis, D. K.; Myslik, J. C.; Salemme, F. R. *Mol. Diversity* **1999**, *4*, 1−22.
(2) Akavia, A. Masters thesis, School of Computer Science, Tel-Aviv University, September 2002.
(3) Balkenhohl, F.; von dem Bussche-Hunnefeld, C.; Lansky, A.; Zechel, C. *Angew. Chem. Int. Ed., Engl.* **1996**, *35*, 2288−2337.
(4) Bayada, D. M.; Hamersma, H.; van Geerestein, V. J. *J. Chem. Inf. Comput. Sci.* **1999**, *39*, 1−10.
(5) Choong, I. C.; Ellman, J. A. *Annu. Rep. Med. Chem.* **1996**, *31*, 309−318.
(6) Cohen, B.; Skiena, S. *J. Comb. Chem.* **2000**, *2*, 10−18.
(7) Corman, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms*, 2nd ed.; MIT Press and McGraw-Hill Book Company: Cambridge, MA, New York, 2001.

(8) Deprez, B.; Williard, X.; Bourel,.L.; Coste, H.; Hyafil, F.; Tartar, A. *Orthogonal combinatorial chemical libraries. J. Am. Chem. Soc.* **1995**, *117*, 5405−5406.
(9) Ellman, J. A. *Acc. Chem. Res.* **1996**, *29*, 132−143.
(10) Cerius^2 Modeling Environment; Release 4.5; Accelrys Inc.: San Diego, 1999.
(11) Fruchtel, J. S.; Jung, G. *Angew. Chem. Int. Ed., Engl.* **1996**, *35* (l), 17−42.
(12) Furka, A.; Christensen, J. W.; Healy, E.; Tanner, H. R.; Saneii, H. *J. Comb. Chem.* **2000**, *2*, 220−223.
(13) Gallop, M. A.; Barrett, R. W.; Dower, W. J.; Fodor, S. P. A.; Gordon, E. M. *J. Med. Chem.* **1994**, *37*, 1233−1251.
(14) Gillet, V. J.; Willett, P.; Bradshaw, J. *J. Chem. Inf. Comput. Sci.* **1997**, *37*, 731−740.
(15) Gordon, E. M.; Barrett, R. W.; Dower, W. J.; Fodor, S. P. A.; Gallop, M. A. *J. Med. Chem.* **1994**, *37*, 1385−1399.
(16) Gordon, E. M.; Gallop, M. A.; Patel, D. V. *Acc. Chem. Res.* **1996**, *29*, 144−154.
(17) Gordon, E. M.; Kerwin, J. F., Jr. *Combinatorial Chemistry and Molecular Diversity in Drug Discovery*; Wiley: New York, 1998.
(18) Hassan, M.; Bielawski, J. P.; Hempel, J. C.; Waldman, M. *Mol. Diversity* **1996**, *2*, 64−74.
(19) Higgs, R. E.; Bemis, K. J.; Watson, I. A.; Wikel, J. H. *J. Chem. Inf. Comput. Sci.* **1997**, *37*, 861−870.
(20) Jung, G. Combinatorial Peptide and Nonpeptide Libraries; VCH: Weinheim, Germany, 1996.
(21) Konings, D. A. M.; Wyatt, J. R.; Ecker, D. J.; Freier, S. M. *J. Med. Chem.* **1997**, *40*, 4386-4395.
(22) Madden, D.; Krchnak, V.; Lebl, M. *Persp. Drug Discovery Des.* **1995**, *29*, 269−285.
(23) Matter, H. *J. Med. Chem.*. **1997**, 1219−1229.
(24) Nefzi, A.; Ostresh, J. M.; Houghten, R. A. *Chem. Rev.* **1997**, *97*, 449−472.
(25) Snarey, M.; Terrett, N. K.; Willett, P.; Wilton, D. J. *J. Mol. Graphics Model.* **1997**, *15*, 372−385.
(26) Thompson, L. A.; Ellman, J. A. *Chem. Rev.* **1996**, *96*, 555−600.
(27) Topiol, S.; Davies, J.; Vijayakuma, S.; Wareing, J. R. Computer aided analysis of ∼ split and mix combinatorial libraries. *J. Comb. Chem.* **2001**, *3*, 20−27.
(28) Zhao, P.; Zambias, R.; Bolognese, J. A.; Boulton, D.; Chapman, K. *Proc. Natl. Acad. Sci., U.S.A.* **1995**, *92*, 10212−10216.

CC034045I