Lexis: An Optimization Framework for Discovering the Hierarchical Structure of Sequential Data

Payam Siyari
GeorgiaTech
Atlanta, GA, USA
payamsiyari@gatech.edu

Bistra Dilkina
GeorgiaTech
Atlanta, GA, USA
bdilkina@cc.gatech.edu

Constantine Dovrolis
GeorgiaTech
Atlanta, GA, USA
constantine@gatech.edu

ABSTRACT

Data represented as strings abounds in biology, linguistics, document mining, web search and many other fields. Such data often have a hierarchical structure, either because they were artificially designed and composed in a hierarchical manner or because there is an underlying evolutionary process that creates repeatedly more complex strings from simpler substrings. We propose a framework, referred to as *Lexis*, that produces an optimized hierarchical representation of a given set of "target" strings. The resulting hierarchy, "Lexis-DAG", shows how to construct each target through the concatenation of intermediate substrings, minimizing the total number of such concatenations or DAG edges. The Lexis optimization problem is related to the smallest grammar problem. After we prove its NP-hardness for two cost formulations, we propose an efficient greedy algorithm for the construction of Lexis-DAGs. We also consider the problem of identifying the set of intermediate nodes (substrings) that collectively form the "core" of a Lexis-DAG, which is important in the analysis of Lexis-DAGs. We show that the Lexis framework can be applied in diverse applications such as optimized synthesis of DNA fragments in genomic libraries, hierarchical structure discovery in protein sequences, dictionary-based text compression, and feature extraction from a set of documents.

1. INTRODUCTION

In both nature and technology, information is often represented in sequential form, as strings of characters from a given alphabet [11]. Such data often exhibit a hierarchical structure in which previously constructed strings are re-used in composing longer strings [21]. In some cases this hierarchy is formed "by design" in synthetic processes where there are some cost savings associated with the re-use of existing modules [15, 19]. In other cases, the hierarchy emerges naturally when there is an underlying evolutionary process that repeatedly creates more complex strings from simpler ones, conserving only those that are being re-used [19, 21].

For instance, language is hierarchically organized starting from phonemes to stems, words, compound words, phrases, and so on [20]. In the biological world, genetic information is also represented sequentially and there is ample evidence that evolution has led to a hierarchical structure in which sequences of DNA bases are first translated into amino acids, then form motifs, regions, domains, and this process continues to create many thousands of distinct proteins [8].

In the context of synthetic design, an important problem is to construct a minimum-cost Directed Acyclic Graph (DAG) that shows how to produce a given set of "target strings" from a given "alphabet" in a hierarchical manner, through the construction of intermediate substrings that are re-used in at least two higher-level strings. The cost of a DAG should be related somehow to the amount of "concatenation work" (to be defined more precisely in the next setion) that the corresponding hierarchy would require. For instance, in *de novo* DNA synthesis [4, 7], biologists aim to construct target DNA sequences by concatenating previously synthesized DNAs in the most cost-efficient manner.

In other contexts, it may be that the target strings were previously constructed through an evolutionary process (not necessarily biological), or that the synthetic process that was followed to create the targets is unknown. Our main premise is that even in those cases it is still useful to construct a cost-minimizing DAG that composes the given set of targets hierarchically, through the creation of intermediate substrings. The resulting DAG shows the most parsimonious way to represent the given targets hierarchically, revealing substrings of different lengths that are highly re-used in the targets and identifying the dependencies between the re-used substrings. Even though it would not be possible to prove that the given targets were actually constructed through the inferred DAG, this optimized DAG can be thought of as a plausible hypothesis for the unknown process that created the given targets as long as we have reasons to believe that that process cares to minimize, even heuristically, the same cost function that the DAG optimization considers. Additionally, even if our goal is not to reverse engineer the process that generated the given targets, the derived DAG can have practical value in the applications such as compression or feature extraction.

In this paper, we propose an optimization framework, referred to as *Lexis*,[1] that designs a minimum-cost hierarchical representation of a given set of target strings. The resulting hierarchy, referred to as "Lexis-DAG", shows how to construct each target through the concatenation of intermediate substrings, which themselves might be the result

---

[1] Lexis means "word" in Greek.

of concatenation of other shorter substrings, all the way to a given alphabet of elementary symbols. We consider two cost functions: minimizing the total number of concatenations and minimizing the number of DAG edges. The choice of cost function is application-specific. The Lexis optimization problem is related to the smallest grammar problem [6, 14]. We show that Lexis is NP-hard for both cost functions, and propose an efficient greedy algorithm for the construction of Lexis-DAGs. Interestingly, the same algorithm can be used for both cost functions. We also consider the problem of identifying the set of intermediate nodes (substrings) that collectively form the "core" of a Lexis-DAG. This is the minimal set of DAG nodes that can cover a given fraction of source-to-target paths, from alphabet symbols to target strings. The core of a Lexis-DAG represents the most central substrings in the corresponding hierarchy. We show that the Lexis framework can be applied in diverse applications such as optimized synthesis of DNA fragments in genomic libraries, hierarchical structure discovery in protein sequences, dictionary-based text compression, and feature extraction from a set of documents.

2. PROBLEM STATEMENT

2.1 Lexis-DAG

Given an alphabet S and a set of "target" strings T over the alphabet S, we need to construct a Lexis-DAG. A Lexis-DAG D is a directed acyclic graph $D(V, E)$, where V is the set of nodes and E the set of edges, that satisfies the following three constraints.[2]

First, each node $v \in V$ in a Lexis-DAG represents a string $\mathcal{S}(v)$ of characters from the alphabet S. The nodes V_S that represent characters of S are referred to as *sources*, and they have zero in-degree. The nodes V_T that represent target strings $T = \{t_1, t_2, \ldots, t_m\}$ are referred to as *targets*, and they have zero out-degree. V also includes a set of *intermediate nodes* V_M, which represent substrings that appear in the targets T. So, $V = V_S \cup V_M \cup V_T$.

Second, each node in $V_M \cup V_T$ of a Lexis-DAG represents a string that is the concatenation of two or more substrings, specified by the incoming edges from other nodes to that node. Specifically, an edge $e \in E$ from node u to node v is a triplet (u, v, i) such that the string $\mathcal{S}(u)$ appears as substring of $\mathcal{S}(v)$ at index i (the first character of a string has index 1). Note that there may be more than one edges from node u to node v. The number of incoming and outgoing edges for v is denoted by $d_{in}(v)$ and $d_{out}(v)$, respectively. $I(v)$ is the sequence of nodes u that appear in the incoming edges (u, v, i) of v, ordered by edge index i. We require that for each node v in $V_M \cup V_T$ replacing the sequence of nodes in $I(v)$ with their corresponding strings results in exactly $\mathcal{S}(v)$.

Third, a Lexis-DAG should only include intermediate nodes that have an out-degree of at least two,

$$\forall v \in V_M, d_{out}(v) \geq 2. \tag{1}$$

In other words, every intermediate node $v \in V_M$ in a Lexis-DAG should be such that the string $\mathcal{S}(v)$ is re-used in at least two concatenation operations. Otherwise, $\mathcal{S}(v)$ is either not used in any concatenation operation, or it is used only once and so the outgoing edge from v can be replaced by re-wiring

---

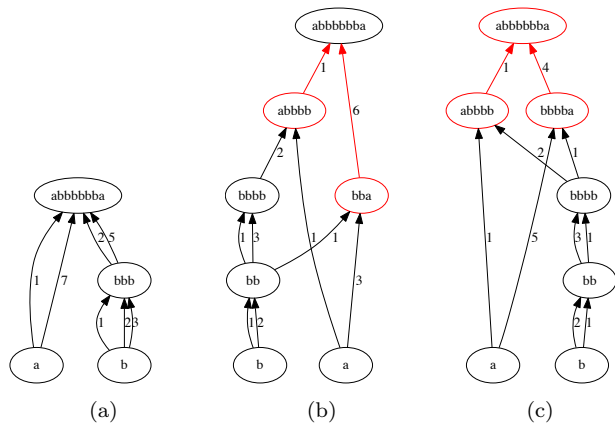[2] To simplify the notation, even though D is a function of S and T, we do not denote it as such.

Figure 1: Illustration of the Lexis-DAG for targets $T = \{abbbbbba\}$ and sources $S = \{a, b\}$. Edge-labels indicate the occurrence indices: **(a)** A valid Lexis-DAG having both minimum number of concatenations and edges. **(b)** An invalid Lexis-DAG: two intermediate nodes are re-used only once. **(c)** An invalid Lexis-DAG: the top-layer string is not equal to the concatenation of its two in-neighbors (best viewed in color).

the incoming edges of v straight to the single occurrence of $\mathcal{S}(v)$. In both cases node v can be removed from the Lexis-DAG, resulting in a more parsimonious hierarchical representation of the targets. Fig. 1 illustrates the concepts introduced in this section.

2.2 The Lexis Optimization Problem

The *Lexis* optimization problem is to construct a minimum-cost Lexis-DAG for the given alphabet S and target strings T. In other words, the problem is to determine the set of intermediate nodes V_M and all required edges E so that the corresponding Lexis-DAG D is optimal in terms of a given cost function $C(D)$.

$$\min_{(E, V_M)} C(D)$$
$$s.t. \ D = (V, E) \text{ is a Lexis-DAG for } S \text{ and } T \tag{2}$$

The selection of an appropriate cost function is somewhat application-specific. A natural cost function to consider is the number of edges in the Lexis-DAG. In certain applications, such as DNA synthesis, the cost is usually measured in terms of the number of required concatenation operations. In the following, we consider both cost functions. Note that we choose to not explicitly minimize the number of intermediate nodes in V_M; minimizing the number of edges or concatenations, however, tends to also reduce the number of required intermediate nodes. Additionally, the constraint (1) means that the optimal Lexis-DAG will not have redundant intermediate nodes that can be easily removed without increasing the concatenation or edge cost. More general cost formulations, such as a variable edge cost or a weighted average of a node cost and an edge cost, are interesting but they are not pursued in this paper.

2.2.1 Edge cost

Suppose that the cost of each edge is one. The *edge cost* to construct a node $v \in V$ is defined as the number of incoming edges required to construct $\mathcal{S}(v)$ from its in-neighbors, which is equal to $d_{in}(v)$. The edge cost of source nodes is obviously zero. The edge cost $\mathcal{E}(D)$ of Lexis-DAG D is defined as the edge cost of all nodes, equal to the number of edges in D:
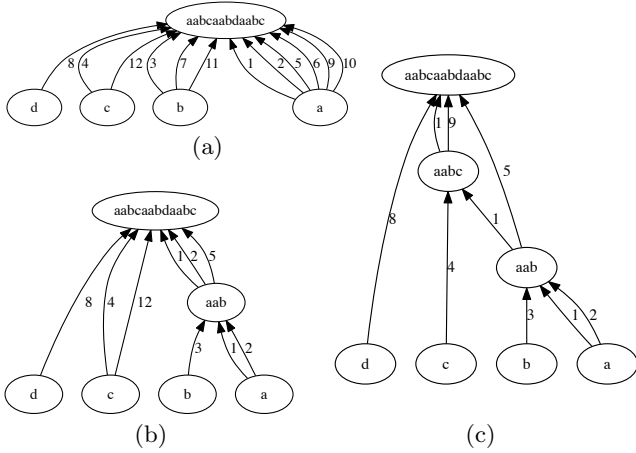
Figure 2: Illustration of G-Lexis given target $T = \{aabcaabdaabc\}$ and sources $S = \{a, b, c, d\}$. - **(a)** Initial Lexis-DAG: The string passed to the suffix tree is $I(aabcaabdaabc) = aabcaabdaabc$. **(b)** Substring aab has maximum *SavedCost*. The target can be now written as $I(aabcaabdaabc) = \sigma_{aab}c\sigma_{aab}d\sigma_{aab}c$ where σ_{aab} is the substring aab of the new intermediate node. We also have that $I(aab) = aab$. The strings passed to the suffix tree are $\{I(aabcaabdaabc), I(aab)\}$. **(c)** Substring $\sigma_{aab}c$ has maximum *SavedCost* and is chosen for a new intermediate node. In this example, this would be the last iteration.

$$\mathcal{E}(D) = \sum_{v \in V} d_{in}(v) = |E| \qquad (3)$$

2.2.2 Concatenation cost

Suppose that the cost of each concatenation operation is one. The *concatenation cost* to construct a node $v \in V_M \cup V_T$ is defined as the number of concatenations required to construct $\mathcal{S}(v)$ from its in-neighbors, which is equal to $d_{in}(v)-1$. The concatenation cost $\mathcal{C}(D)$ of Lexis-DAG D is defined as the concatenation cost of all non-source nodes; it is easy to see that this is equal to the number of edges in D minus the number of non-source nodes,

$$\mathcal{C}(D) = \sum_{v \in V \setminus V_S} (d_{in}(v) - 1) = |E| - |V \setminus V_S| \quad (4)$$

THEOREM 1. *The optimization problem in Eq. (2) is NP-hard for both the edge cost of Eq. (3) and the concatenation cost of Eq. (4).*

The proof is given in the Appendix. Note that the objective in Eq. (4) is an explicit function of the number of intermediate nodes in the Lexis-DAG. Hence the optimal solutions for the concatenation cost can be different than those for the edge cost. An example is shown in the Appendix.

3. THE GREEDY LEXIS ALGORITHM

In this section, we describe a greedy algorithm, referred to as G-LEXIS, for both previous optimization problems. The basic idea in G-LEXIS is that it searches for the substring ξ that will lead, under certain assumptions, to the maximum cost reduction when added as a new intermediate node in the Lexis-DAG. The algorithm starts from the trivial Lexis-DAG with no intermediate nodes and edges from the source nodes representing alphabet symbols to each of their occurrences in the target strings.

Recall that for every node $v \in V_T \cup V_M$, $I(v)$ is the sequence of nodes appearing in the incoming edges of v, i.e., the sequence of nodes whose string concatenation results in the string $\mathcal{S}(v)$ represented by v. The sequences $I(v)$ can be interpreted as strings over the "alphabet" of Lexis-DAG nodes. Note that every symbol in a string $I(v)$ has a corresponding edge in the Lexis-DAG. We look for a repeated substring ξ in the strings $I_{T\cup M} = \{I(v)|v \in V_T \cup V_M\}$ that can be used to construct a new intermediate node. We can construct a new intermediate node for ξ, create incoming edges based on the symbols in ξ (remember ξ is a substring over the alphabet of nodes), and replace the incoming edges to each of the non-overlapping repeated occurrences of ξ with a single outgoing edge from the new node.

Consider the edge cost first. Suppose that ξ is repeated $R_{T\cup M,\xi}$ times in the strings $I_{T\cup M}$. If these occurrences of ξ are non-overlapping, the number of required edges would be $|\xi| R_{T\cup M,\xi}$. After we construct a new intermediate node for ξ as outlined above, the edge cost will be $|\xi| + R_{T\cup M,\xi}$. So, the reduction in edge cost from re-using ξ would be $(R_{T\cup M,\xi} - 1)(|\xi| - 1) - 1$. Under the stated assumptions about ξ, this reduction is non-negative if ξ is repeated at least twice and its length is at least two.

Consider the concatenation cost now. If these occurrences of ξ are non-overlapping, the number of required concatenations for all the repeated occurrences would be $(|\xi| - 1) R_{T\cup M,\xi}$. After we construct a new intermediate node for ξ as outlined above, the concatenation cost will be $|\xi| - 1$. We expect a reduction in the number of required concatenations by $(R_{T\cup M,\xi} - 1)(|\xi| - 1)$.

So, the greedy choice for both cost functions is the same: select the substring ξ that maximizes the term $SavedCost = (R_{T\cup M,\xi} - 1)(|\xi| - 1)$. For this reason, our G-LEXIS algorithm can be used for both cost functions we consider. It starts with the trivial Lexis-DAG, and at each iteration it chooses a substring of $I_{T\cup M}$ in the Lexis-DAG that maximizes $SavedCost$, creates a new intermediate node for that substring and updates the edges of the Lexis-DAG accordingly. The algorithm terminates when there are no more substrings of $I_{T\cup M}$ with length at least two and repeated at least twice. The pseudocode for G-LEXIS is shown in Algorithm 1. An example of application of the G-LEXIS algorithm is shown in Fig. 2.

ALGORITHM 1. G-LEXIS

Input: Alphabet S, Targets T **Output:** Lexis-DAG D

1. Initialize $V \leftarrow V_T \cup V_S$ and E, constructing each target in T from characters in S. $V_M \leftarrow \varnothing$.
2. Repeat:
   (a) $I_{T\cup M} \leftarrow \{I(v)|v \in (V_T \cup V_M)\}$;
   (b) Select ξ with maximum $(R_{T\cup M,\xi} - 1)(|\xi| - 1)$, where $R_{T\cup M,\xi}$ is the number of repeats of substring ξ in $I_{T\cup M}$; #GreedyChoice
   (c) If $(R_{T\cup M,\xi} - 1)(|\xi| - 1) = 0$, *break;*
   (d) $V \leftarrow V \cup \{\sigma_\xi\}$, where σ_ξ is the new intermediate node, and update E accordingly; #UpdateLexis-DAG

At each iteration of G-LEXIS, we need to find efficiently the substring of $I_{T\cup M}$ with maximum *SavedCost*. We observe that the substring that maximizes *SavedCost* is a "maximal repeat". Maximal repeats are substrings of length at least two, whose extension to the right or left would reduce

its occurrences in the given set of strings. Suppose that it is not. Then, there is a substring $\hat{\sigma}$, which is not a maximal repeat, that maximizes *SavedCost*. If we can extend $\hat{\sigma}$ to the left or right we can increase its length without reducing its number of occurrences. By doing so, we construct a new substring with higher *SavedCost* than $\hat{\sigma}$, violating our initial assumption. So, the substring that maximizes *SavedCost* is a maximal repeat. A suffix tree over a set of input strings captures all right-maximal repeats, and right-maximal repeats are a superset of all maximal repeats [11]. To pick the one with maximum *SavedCost*, we need the count of non-overlapping occurrences of these substrings. A Minimal Augmented Suffix Tree [5] over $I_{T \cup M}$ can be constructed and used to count the number of non-overlapping occurrences of all right-maximal repeats in overall $O(L \log L)$ time, where L is the total length of target strings. Using a regular suffix tree instead, this can be achieved in only $O(L)$ time; but suffix tree may count overlapping occurrences. In our implementation we prefer to use regular suffix tree, following related work [10] that has shown that this performance optimization has negligible impact on the solution's quality. So, the substring that is chosen for the new Lexis-DAG node is based on length and overlapping occurrence count. We then use the suffix tree to iterate over all occurrences of the selected substring, skipping overlapping occurrences. If a selected substring has less than two non-overlapping occurrences, we skip to the next best substring. Using the suffix tree, we can update the Lexis-DAG with the new intermediate node, and with the corresponding edges for all occurrences of that substring, in $O(L)$ time. The maximum number of iterations of G-LEXIS is $O(L)$ because each iteration reduces the number of edges (or concatenations), which at the start is $O(L)$. So, the overall run-time complexity using suffix tree is $O(L^2)$.

We have also experimented with other algorithms, such as a greedy heuristic that selects the longest repeat in each iteration of building the DAG, i.e., it chooses based on length among all substrings that appear at least twice in the targets or intermediate node strings. This heuristic can be efficiently implemented to run in only $O(L)$ time [13]. Our evaluation shows that G-LEXIS performs significantly better than the longest repeat heuristic in terms of solution quality, despite some running time overhead. Running both algorithms on a machine with an Intel Core-i7 2.9 GHz CPU and 16GB of RAM on the NSF abstracts dataset (introduced in Section 5) of $2,309$ target strings with total length $245,968$ symbols takes 562 sec for G-LEXIS and 408 sec for the longest repeat algorithm. The edge cost with G-LEXIS is 169,060 compared to 183,961 with the longest repeat algorithm. More detailed results can be found in the Appendix.

4. PATH-CENTRALITY AND THE CORE OF A LEXIS-DAG

After constructing a Lexis-DAG, an important question is to rank the constructed intermediate nodes in terms of significance or *centrality*. Even though there are many related metrics in the network analysis literature, such as closeness, betweenness or eigenvector centrality [22], none of them captures well the semantics of a Lexis-DAG. In a Lexis-DAG, a path that starts from a source and terminates at a target represents a dependency chain in which each node depends on all previous nodes in that path. So, the higher the number of such source-to-target paths traversing an intermediate

node v is, the more important v is in terms of the number of dependency chains it participates in. More formally, let $P_D(v)$ be the number of source-to-target paths that traverse node $v \in V_M$; we refer to $P_D(v)$ as the *path centrality* of intermediate node v. The path centrality of sources and targets is zero by definition. First, note that:

$$P(v) = P_S(v)\, P_T(v) \qquad (5)$$

where $P_S(v)$ is the number of paths from any source to v, and $P_T(v)$ is the number of paths from v to any target. This suggests an efficient way to calculate the path centrality of all nodes in a Lexis-DAG in $O(|E|)$ time: perform two DFS traversals, one starting from sources and following the direction of edges, and another starting from targets and following the opposite direction. The first DFS traversal will recursively produce $P_S(v)$ while the second will produce $P_T(v)$, for all intermediate nodes.

Second, it is easy to see that $P_T(v)$ is equal to the number of times string $\mathcal{S}(v)$ is used for replacement in the target strings T. Similarly, $P_S(v)$ is equal to the number of times any source node is repeated in $\mathcal{S}(v)$, which is simply the length of $\mathcal{S}(v)$. So, the path centrality $P(v)$ of a node in a Lexis-DAG can be also interpreted as its "re-use count" (or number of replaced occurrences in the targets) times its length. Thus, an intermediate node will rank highly in terms of path centrality if it is both long and frequently re-used.

An important follow-up question is to identify the *core* of a Lexis-DAG, i.e., a set of intermediate nodes that represent, as a whole, the most important substrings in that Lexis-DAG. Intuitively, we expect that the core should include nodes of high path centrality, and that almost all source-to-target dependency chains of the Lexis-DAG should traverse at least one of these core nodes.

More formally, suppose K is a set of intermediate nodes and $\mathcal{P}^-(K)$ is the set of source-to-target paths after we remove the nodes in K from D. The core of D is defined as the minimum-cardinality set of intermediate nodes $\hat{K}$ such that the fraction of remaining source-to-target paths after the removal of $\hat{K}$ is at most τ:

$$\min_{K \subseteq V_M} \quad |K|$$
$$s.t. \ |\mathcal{P}^-(K)| \leq \tau\, |\mathcal{P}^-(\varnothing)| \qquad (6)$$

where $|\mathcal{P}^-(\varnothing)|$ is the number of source-to-target paths in the original Lexis-DAG, without removing any nodes.[3]

Note that if $\tau = 0$ the core identification problem becomes equivalent to finding the min-vertex-cut of the given Lexis-DAG. In practice a Lexis-DAG often includes some *tendril-like* source-to-target paths traversing a small number of intermediate nodes that very few other paths traverse. These paths can cause a large increase in the size of the core. For this reason, we prefer to consider the case of a positive, but potentially small, value of the threshold τ.

We solve the core identification problem with a greedy algorithm referred to as G-CORE. This algorithm adds in each iteration the node with the highest path-centrality value to the core set, updates the Lexis-DAG by removing that node and its edges, and recomputes the path centralities before the next iteration. The algorithm terminates when the desired fraction of source-to-target paths has been achieved. G-CORE requires at most $O(|V|)$ iterations, and in each it-

---

[3]It is easy to see that $|\mathcal{P}^-(\varnothing)|$ is equal to the cumulative length of all target strings L.

eration we update the path centralities in $O(|E|)$ time. So the run-time complexity of G-CORE is $O(|V||E|)$.

5. APPLICATIONS OF LEXIS

We now discuss a variety of applications of the proposed framework. Note that in all experiments, we use the library from [10] for extracting the maximal repeats and NetworkX [12] as the graph library in our implementation.

5.1 Optimized String Hierarchies

Lexis can be used as an optimization tool for the hierarchical synthesis of sequences. One such application comes from synthetic biology, where novel DNA sequences are created by concatenating existing DNA sequences in a hierarchical process [7]. The cost of DNA synthesis is considerable today due to the biochemical operations that are required to perform this "genetic merging" [7, 4]. Hence, it is desirable to re-use existing DNA sequences, and more generally, to perform this design process in an efficient hierarchical manner.

Biologists have created a library of synthetic DNA sequences, referred to as iGEM [2]. Currently, there are 787 elementary "BioBrick parts" from which longer composite sequences can be created. Longer sequences are submitted to the Registry of Standard Biological Parts in the annual iGEM competition, then functionally evaluated and labeled. In the following, we analyze a subset of the iGEM dataset. In particular, this dataset contains $1,375$ composite DNA sequences that are labeled as iGEM *devices* because they have distinct biological functions; we treat these sequences as Lexis targets. The cumulative length of the target sequences is $6,957$ symbols. The 787 elementary BioBrick parts are treated as the Lexis sources. The iGEM dataset also includes other BioBrick parts that are neither devices nor elementary, and that have been used to construct more complex parts in iGEM; we ignore those because they do not have a distinct biological function (i.e., they should not be viewed as targets but as intermediate sequences that different teams of biologists have previously constructed).

We constructed an optimized Lexis-DAG for the given sets of iGEM sources and targets. To quantify the gain that results from using a hierarchical synthesis process, we compare the number of edges and concatenations in the Lexis-DAG versus a flat synthesis process in which each target is independently constructed from the required sources. The *Lexis* solution requires only 52% of the edges (or 56% of the concatenations) that the flat process would require. The sequence with the highest path centrality in the Lexis-DAG is *B0010-B0012*.[4] This part is registered as *B0015* in the iGEM library and it is the most common "terminator" in iGEM devices. *Lexis* identified several more high centrality parts that are already in iGEM, such as *B0032-E0040-B0010-B0012*, registered as *E0240*. Interestingly, however, the Lexis-DAG also includes some high centrality parts that have not been registered in iGEM yet, such as *B0034-C0062-B0010-B0012-R0062*. A list of the top-15 nodes in terms of path centrality is given in the Appendix.

To explore the hierarchical nature of the iGEM sequences, we compared the *"Original"* Lexis-DAG, the one we constructed with the actual iGEM devices as targets, with *"Randomized"* Lexis-DAGs. *"Randomized"* Lexis-DAG is the result of applying G-LEXIS to a target set where each iGEM

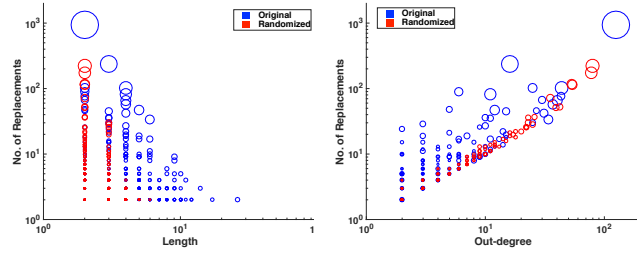[4] BioBricks start with *BBa_* prefix that are omitted here.

Figure 3: Comparison of the Lexis-DAGs that result from the Original and Randomized iGEM devices (best viewed in color). The size of each point represents the path-centrality of that node.

device sequence is randomly reshuffled. We compare the Original Lexis-DAG characteristics to the average Lexis-DAG characteristics over ten randomized experiments. The Original Lexis-DAG has fewer intermediate nodes than the Randomized ones (169 in Original vs 359 in Randomized), and its depth is twice as large (8 vs 4.4). Importantly, the Randomized DAGs are significantly more costly: 44% higher cost in terms of edges and 52% in terms of concatenations.

To further understand these differences from the topological perspective, Fig. 3 shows scatter plots for the length, path centrality, and re-use (number of replacements) of each intermediate node in the Original Lexis-DAG vs one of the Randomized Lexis-DAGs. With randomized targets, the intermediate nodes are short (mostly 2-3 symbols), their re-use is roughly equal to their out-degree, and their path centrality is determined by their out-degree; in other words, most intermediate nodes are directly connected to the targets that include them, and the most central nodes are those that have the highest number of such edges. On the contrary, with the original targets we find longer intermediate nodes (up to 11-12 symbols) and their number of replacements in the targets can be up to an order of magnitude higher than their out-degree. This happens when intermediate nodes with a large number of replacements are not only used directly to construct targets but they are repeatedly combined to construct longer intermediate nodes, creating a deeper hierarchy of re-use. In this case, the high path centrality nodes tend to be those that are both relatively long and common, achieving a good trade-off between specificity and generality.

5.2 Structure Discovery

As mentioned in the introduction, it is often the case that the hierarchical process that creates the observed sequences is unknown. *Lexis* can be used to discover underlying hierarchical structure as long as we have reasons to believe that that hierarchical process cares to minimize, even heuristically, the same cost function that Lexis considers (i.e., number of edges or concatenations). A related reason to apply Lexis in the analysis of sequential data is to identify the most parsimonious way, in terms of number of edges or concatenations, to represent the given sequences hierarchically. Even though this representation may not be related to the process that generated the given targets, it can expose if the given data have an inherent hierarchical structure.

As an illustration of this process, we apply Lexis on a set of protein sequences. Even though it is well-known that such sequences include conserved and repeated subsequences (such as motifs) of various lengths, it is not currently known whether these repeats form a hierarchical structure. That

would be the case if one or more short conserved sequences are often combined to form longer conserved sequences, which can themselves be combined with others to form even longer sequences, etc. If we accept the premise that a conserved sequence serves a distinct biological function, the discovery of hierarchical structure in protein sequences would suggest that elementary biological functions are combined in a Lego-like manner to construct the complexity and diversity of the proteome. In other words, the presence of hierarchical structure would suggest that proteins satisfy, at least to a certain extent, the *composability* principle, meaning that the function of each protein is composed of, and it can be understood through, the simpler functions of hierarchical components.

Our dataset is the proteome of baker's Yeast[5], which consists of 6,721 proteins. However, this includes many protein homologues. It is important that we replace each cluster of homologues with a single protein; otherwise Lexis can detect repeated sequences within two or more homologues. To remedy this issue, we use the *UCLUST* sequence clustering tool [1], which is based on the *USEARCH* similarity measure (or identity search) [9]. The Percentage of Identity (PID) parameter controls how similar two sequences should be so that they are assigned to the same cluster. We set PID to 50%, which reduces the number of proteins to 6,033. Much higher PID values do not cluster together some obvious homologues, while lower PID values are too restrictive.[6] To reduce the running time associated with the randomization experiments described next, we randomly sample 1,500 proteins from the output of *UCLUST*.

The total length of the protein targets is about 344K amino acids. The resulting Lexis-DAG has about 151K edges and 5,171 intermediate nodes, and its maximum depth is 7. Fig. 4(a) shows a scatter plot of the length and number of replacements of these intermediate Lexis nodes (repeated sequences discovered by Lexis).

Of course some of these sequences may not have any biological significance because their length and number of replacements may be so low that they are likely to occur just based on chance. For instance, a sequence of two amino acids that is repeated just twice in a sequence of thousands of amino acids is almost certain (the distribution of amino acids is not very skewed). To filter out the sequences that are *not* statistically significant, we rely on the following hypothesis test. Consider a node that corresponds to a sequence with length l and number of replacements r in the given targets. The null-hypothesis is that sequences with these values of l and r will occur in a Lexis-DAG that is constructed for a random permutation of the given targets. To evaluate this hypothesis, we randomize the given target sequences multiple times, and construct a Lexis-DAG for each randomized sequence. We then estimate the probability that sequences of length l and number of replacements r occur in the randomized target Lexis-DAG, as the fraction of 500 experiments in which this is true. For a given significance level $\alpha = 0.1$, we can then identify the pairs (l, r) for which we can reject the null-hypothesis; these pairs correspond to the nodes that we view as statistically significant.[7]

On average, the randomized target Lexis-DAGs have a smaller depth (5.0) and more edges ($155K$) than the original Lexis-DAG. Fig. 4(b) shows the intermediate nodes of the original Lexis-DAG that pass the previous hypothesis test.
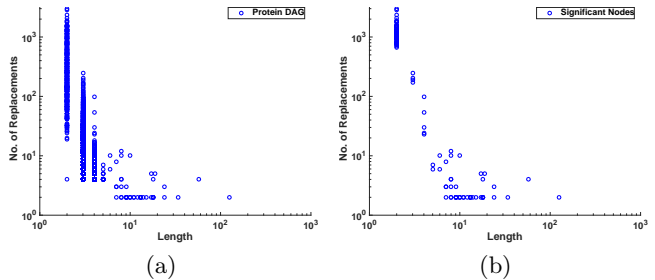

(a)   (b)

Figure 4: Length and number of replacements for the intermediate nodes in the protein sequence Lexis-DAG (a) before and (b) after we filter out the nodes that are not statistically significant.

Fig. 5 shows a small subgraph of the Lexis-DAG, showing only about 30 intermediate nodes; all these nodes have passed the previous significance test. The grey lines represent indirect connections, going through nodes that have not passed the significance test (not shown), while the bold lines represent direct connections. Interestingly, there seems to be a non-trivial hierarchical structure with several long paths, and with sequences of several amino acids that repeat several times even in this relatively small sample of proteins. Despite these preliminary results, it is clearly still early to draw any hard conclusions about the presence of hierarchical structure in protein sequences. We are planning to further pursue this question using Lexis in collaboration with domain experts.

5.3 Compression

Recent work has highlighted the connection between pattern mining and sequence compressibility [16]. Data compression looks for regularities that can be used to compress the data, while patterns are often useful as such regularities. In dictionary-based lossless compression, the original sequence is encoded with the help of a dictionary, which is a subset of the sequence's substrings as well as a series of pointers indicating the location(s) at which each dictionary element should be placed to fully reconstruct the original sequence. Following the Minimum Description Principle, one strives for a compression scheme that results in the smallest size for the joint representation of both the dictionary and the encoding of the data using that dictionary. The size of this joint representation is the total space needed to store the dictionary entries plus the total space needed for the required pointers. We assume for simplicity that the space to store an individual character and a pointer are the same.

We now evaluate the use of a Lexis-DAG for compression (or compact representation) of strings. To do so, we need to decide 1) how to choose the patterns that will be used for compression, and 2) if a pattern appears more than once, which occurrences of that pattern to replace. A naive approach is to simply use the set of substrings that appear in the Lexis-DAG as dictionary entries, and compare them to sets of patterns found by other substring mining algorithms.

---

[5]http://www.uniprot.org/proteomes/UP000002311

[6]http://drive5.com/usearch/manual/uclust_algo.html

[7]Another way to conduct this hypothesis test would be to estimate the probability that a specific sequence of length l will be repeated r times in a permutation of the targets. The number of randomization experiments would need to

be much higher in that case, however, to cover all sequences that we see in the actual Lexis-DAG, each with a given value of r.
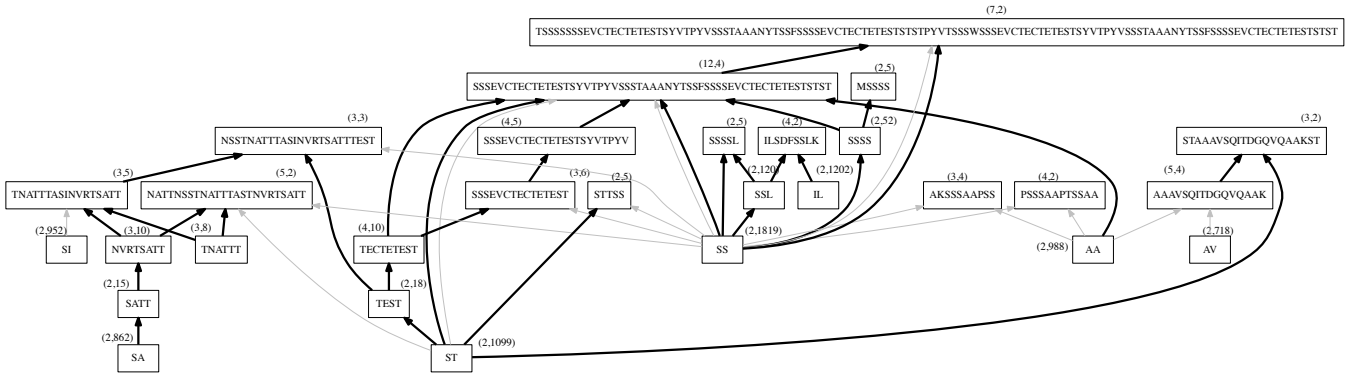
Figure 5: A small subgraph of the Lexis-DAG for 1,500 Yeast proteins. This plot only shows statistically significant nodes. Bold edges represent a direct connection between two nodes, while grey edges represent indirect connections (through nodes that are not shown in this plot). The label on top of each node shows the tuple (in-degree, number of replacements).

Given a set of patterns as potential dictionary entries, selecting the best dictionary and pointer placement is NP-hard. A simple greedy compression scheme, that we refer to as *CompressLR*, is to iteratively add to the dictionary the substring that gives the highest compression gain when replacing all Left-to-Right non-overlapping occurrences of that substring with pointers. We re-evaluate the compression gain of candidate dictionary entries in each iteration. For a substring v with length $|v|$ and number of left-to-right non-overlapping occurrences R_v, the compression gain is:

$$R_v \times |v| - R_v - |v| = (R_v - 1) \times (|v| - 1) - 1 \qquad (7)$$

We compare the substrings identified by Lexis with the substrings generated by a recent contiguous pattern mining algorithm called *ConSgen* [27] (we could only run it on the smallest available dataset). Additionally, we compare the *Lexis* substrings with the set of patterns containing all 2- and 3-grams of the data. The comparisons are performed on six sequence datasets: the "Yeast" and iGEM datasets of the previous sections, as well as four "NSF CS awards" datasets that will be described in more detail in the next section.

Table 1 shows the comparison results under the headings: Lexis-CompressLR, 2+3grams-CompressLR and ConSgen-CompressLR. These naive approaches are all on par with each other. This comparison, however, treats G-Lexis as a mere pattern mining algorithm. Instead, the G-Lexis algorithm constructs a Lexis-DAG that puts the generated patterns in a hierarchical context. One can think of the Lexis-DAG as the instructions in constructing a hierarchical "Lego-like" sequence. The edges into the targets tell us how to place the final pointers, i.e., which occurrences of a dictionary entry to replace in the targets. Further, the rest of the DAG shows how to compress the patterns that appear in the targets using smaller patterns. It is easy to see that using this strategy the compressed size becomes equal to the number of edges in the DAG. Using this strategy that is encoded in the Lexis-DAG results in an additional 2%-20% reduction in the compressed size over the CompressLR approaches.

5.4 Feature Extraction

The Lexis-DAG can also be used to extract machine learning features for sequential data. The intermediate nodes that form the core of a Lexis-DAG, in particular, correspond to sequences that are both relatively long and frequently re-

Dataset	Lexis DAG	Lexis CompressLR	2+3-gram CompressLR	ConSgen CompressLR
Know&Cog	**68.69**	76.58	77.13	—
Networks	**78.48**	86.47	86.43	—
Robotics	**73.19**	80.62	79.69	—
Theory	**79.41**	81.89	82.63	—
Yeast	**44.28**	51.08	50.71	—
iGEM	**47.86**	67.47	67.75	67.47

Table 1: Compression ratio (i.e., percentage of compressed data size over original data size).

used in the targets. We hypothesize that such sequences will be good features for machine learning tasks such as classification or clustering because they can discriminate different classes of objects (due to their longer length) and at the same time are general within the same class of objects (due to their frequent occurrence in the targets of that class).

To test this hypothesis, we used Lexis to extract text features for four classes of NSF research award abstracts during the 1990–2003 time period.[8] We pre-processed each award's abstract through stopword removal and Porter stemming. The alphabet S is the set of word stems that appear at least once in any of these abstracts. Table 2 describes this dataset in terms of number of abstracts, cumulative abstract length, and average length per abstract for each class.

Table 2: Description of 4 classes of NSF award abstracts

Class	$|T|$	L	$L/|T|$	$|V_M|$
Knowledge &Cog Sci	411	47,858	116	2,902
Networks	836	74,738	89	3,730
Robotics	496	56,481	113	4,560
Theory	566	66,891	118	4,247

We constructed the Lexis-DAG for each class of abstracts, and then used the G-Core algorithm to identify the core for each DAG. We stopped G-Core at the point where 95% of indirect paths in the Lexis-DAG are covered. The strings in each core are the extracted features for the corresponding class of abstracts. Table 3 shows the 5 strings extracted by G-Core for each class. We create a common set of G-Core features by taking the union of the sets of core substrings derived for each class. The next step is to construct the

---

[8]archive.ics.uci.edu/ml/machine-learning-databases/nsfabs-mld/nsfawards.data.html

feature vector for each abstract. We do so by representing each abstract as a vector of counts, with a count for each substring feature.

Table 3: 5 word stems in the Lexis-DAG core of each class of abstracts.

Knowledge & Cog	Networks
machine learn	request support nsf connect
knowledg base	bit per second
natur languag	two year
artifici intellig	provide partial support
neural network	high perform
Robotics	**Theory**
comput vision	comput scienc
first year	real world
robot system	complex class
object recognit	complex theori
real time	approxim algorithm

Table 4: SVM classification accuracy. # nonzeros is the number of nonzero elements in the term-document matrix with each feature set. The accuracies and the parameters are fit based on 10-fold cross-validation for each feature set.

Method (# Features)	# Nonzeros	(γ, c)	Accuracy
Bag-of-words (9,7k)	190,2k	*(0.0015,3)*	88.3%
G-Core (14,4k)	**55,5k**	*(0.02,1)*	90.0%
2-Gram (124,9k)	228,0k	*(0.0015,3)*	**91.3%**
3-Gram (186,3k)	234,5k	*(0.001,1)*	75.8%
1+2-Gram (134,6k)	418,2k	*(0.001,1)*	90.9%
1+2+3-Gram (321,0k)	652,8k	*(0.001,1)*	89.2%

To assess how good these features are, we compare the classification accuracy obtained using the Lexis features with more mainstream representations in text mining on NSF data: "bag-of-words", 2-gram, 3-gram, and two combinations of these representations. We use a basic SVM classifier with an RBF kernel. We used the SVM implementation in MATLAB. The accuracy results are similar to those with a KNN classifier that we tried with a Cosine distance, and the accuracy is evaluated with 10-fold cross-validation.

Table 4 shows that the Lexis features result in a much sparser term-document matrix, and so in smaller data overhead in learning tasks, without sacrificing classification accuracy. Lexis also results in a lower feature dimensionality (with the exception of the 1-gram method but the accuracy of that method is much lower). Note that most *Lexis* features are 2-grams but the *Lexis* core (for 95% of path coverage) may also include longer n-grams. Lexis becomes better, relative to the other feature sets, as we decrease the number of considered features. For instance, with 3,000 features the accuracy with Lexis is 74%, while the accuracy with the 1-gram and 2-gram features is 69% and 64%, respectively.

6. RELATED WORK

Lexis is closely related to the *Smallest Grammar Problem* (SGP), which focuses on the following question: *What is the smallest context-free grammar that only generates a given string?* The constraint that the grammar should generate only one string is important because otherwise we could simply consider a Σ^* as the generator of any string over Σ. The SGP is NP-hard, and inapproximable beyond a ratio of $\frac{8569}{8568}$ [6]. Algorithms for SGP have been used for string compression [18] and structure discovery [21] (for a survey see [10]). There are major differences between *Lexis* and SGP. First, in *Lexis* we are given a set of several target strings, not only

one string. Second, *Lexis* infers a network representation (a DAG) instead of a grammar, and so it is a natural tool for questions relating to the hierarchical structure of the targets. For instance, the centrality analysis of intermediate nodes or the core identification question are well understood problems in network analysis, while it is not obvious how to approach them with a grammar-based approach.

One can also relate *Lexis* to the body of work on *sequence pattern mining*, where one is interested in discovering frequent or interesting patterns in a given sequence. Most work in this area has focused on mining subsequences, i.e., a set of ordered but not necessarily adjacent characters from the given sequence. In Lexis, we focus on identifying substrings, also known as *contiguous sequence patterns*. A couple of recent papers develop algorithms for mining substring patterns [27, 26], since sequence mining algorithms do not readily apply to the contiguous case. However, they rely on the methodology of candidate generation (commonly used in sequence pattern mining), where all patterns meeting a certain criterion are found, such as having frequency of at least two or being maximal. In the sequence mining literature, it has been recently observed that the size of the discovered set of patterns as well as their redundancy can be better controlled by mining for a set of patterns that meet a criterion collectively, as opposed to individually. This is useful when these patterns are used as features in other tasks such as summarization or classification. Algorithms for such set-based pattern discovery have been recently developed for sequence pattern mining [16, 25]. In the context of substring pattern mining, Paskov et al [24] show how to identify a set of patterns with optimal lossless compression cost in an unsupervised setting, to be then used as features in supervised learning for classification. In a follow-up paper [23], DRACULA provides a "deep variant" of [24] that is similar to Lexis, in terms of the general problem setup. DRACULA's focus is mostly on complexity and learning aspects of the problem, while Lexis focuses on network analysis of the resulting optimized hierarchy. For instance, DRACULA considers how to take into account how dictionary strings are constructed to regularize learning problems, and how the optimal Dracula solution behaves as the cost varies. We have shown that, although not specifically designed for feature extraction or compression, the *Lexis* framework also results in a small and non-redundant set of substring patterns that can be used in classification and compression tasks.

Optimal DNA synthesis is a new application domain, and we are only aware of the work by Blakes et. al. [4]; they describe DNALD, an algorithm that greedily attempts to maximize DNA re-use for multistage assembly of DNA libraries with shared intermediates. Even though the *Lexis* framework was not specifically designed for DNA synthesis, the Lexis-DAGs can be seemlessly used as solutions for this task. In our illustrative example with the iGEM dataset, G-Lexis returns solutions with 11% lower synthesis cost (equivalent to concatenation cost) than DNALD.

Structure discovery in strings has been explored from several different perspectives. For example, the grammar-based algorithm SEQUITIR [21] presents interesting possible applications in natural language and musical structure identification. In an information-theoretic context, Lancot et. al. [17] shows how to distinguish between coding and non-coding regions by analyzing the hierarchical structure of genomic sequences.

7. CONCLUSIONS AND FUTURE WORK

Lexis is a novel optimization-based framework for exploring the hierarchical nature of sequence data. In this paper, we stated the corresponding optimization problems in the admittedly limited context of two simple cost functions (number of edges and concatenations), proved their NP-hardness, and proposed a greedy algorithm for the construction of Lexis-DAGs. This research can be extended in the direction of more general cost formulations and more efficient algorithms. Additionally, we are working on an incremental version of Lexis in which new targets are added to an existing Lexis-DAG, without re-designing the hierarchy.

We also applied network analysis in Lexis-DAGs, proposing a new centrality metric that can be used to identify the most important intermediate nodes, corresponding to substrings that are both relatively long and frequently occurring in the target sequences. This network analysis connection raises an interesting question: are there certain topological properties that are common across Lexis-DAGs that have resulted from long-running evolutionary processes? We have some evidence that one such topological property is that these DAGs exhibit the *hourglass effect* [3].

Finally, we gave four examples of how Lexis can be used in practice, applied in optimized hierarchical synthesis, structure discovery, compression and feature extraction. In future work, we plan to apply Lexis in various domain-specific problems. In biology, in particular, we can use Lexis in comparing the hierarchical structure of protein sequences between healthy and cancerous tissues. Another related application is *generalized phylogeny inference*, considering that horizontal gene transfers (which are common in viruses and bacteria) result in DAG-like phylogenies (as opposed to trees).

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] *drive5.com/usearch/manual/uclust_algo.html*.

[2] *igem.org/main_page*.

[3] S. Akhshabi and C. Dovrolis. The evolution of layered protocol stacks leads to an hourglass-shaped architecture. In *Dynamics On and Of Complex Networks, Volume 2*. Springer, 2013.

[4] J. Blakes, O. Raz, U. Feige, J. Bacardit, P. Widera, T. Ben-Yehezkel, E. Shapiro, and N. Krasnogor. Heuristic for maximizing DNA re-use in synthetic DNA library assembly. *ACS Synthetic Biology*, 3(8):529–542, 2014.

[5] G.S. Brodal, R.B. Lyngso, A. Ostlin, and C.N.S. Pedersen. Solving the string statistics problem in time $o(n \log n)$. In *ICALP*, pages 728–739, 2002.

[6] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The Smallest Grammar Problem. *IEEE Trans. on Inf. Theory*, 51(7), 2005.

[7] D. Densmore, T.H. Hsiau, J.T. Kittleson, W. DeLoache, C. Batten, and J.C. Anderson. Algorithms for automated DNA assembly. *Nucleic Acids Res.*, 38(8):2607–2616, 2010.

[8] W. Dubitzky, M. Granzow, and D. P. Berrar. *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media, 2007.

[9] R. C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, 2010.

[10] M. Gallé. *Searching for compact hierarchical structures in DNA by means of the Smallest Grammar Problem*. PhD thesis, Université Rennes 1, 2011.

[11] D. Gusfield. Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge Univ. Press, 1997.

[12] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, 2008.

[13] S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara. Linear-time off-line text compression by longest-first substitution. In *String Processing and Information Retrieval*, pages 137–152, 2003.

[14] J.C. Kieffer and E. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. on Inf. Theory*, 46:2000, 2000.

[15] J. Kleinberg. Bursty and hierarchical structure in streams. In *KDD*, pages 91–101, 2002.

[16] H. T. Lam, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. *Statistical Analysis and Data Mining*, 7(1):34–52, 2014.

[17] J.K. Lanctot, M. Li, and E. Yang. Estimating DNA sequence entropy. In *ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[18] N.J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference, 1999. Proceedings.*, pages 296–305, 1999.

[19] H. Mengistu, J. Huizinga, J.B. Mouret, and J. Clune. The evolutionary origins of hierarchy. *CoRR*, abs/1505.06353, 2015.

[20] S. Miyagawa, R.C. Berwick, and K. Okanoya. The emergence of hierarchical structure in human language. *Frontiers in Psychology*, 4(71), 2013.

[21] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, 1997.

[22] M. Newman. *Networks: An introduction*. Oxford University Press, Inc., 2010.

[23] H. S. Paskov, J. C. Mitchell, and T. J. Hastie. Data representation and compression using linear-programming approximations. In *ICLR*, 2016.

[24] H. S. Paskov, R. West, J. C. Mitchell, and T. Hastie. Compressive feature learning. In *NIPS*, pages 2931–2939, 2013.

[25] N. Tatti and J. Vreeken. The long and the short of it: Summarising event sequences with serial episodes. In *KDD*, pages 462–470, 2012.

[26] J. Zhang, Y. Wang, and D. Yang. CCSpan: Mining closed contiguous sequential patterns. *Knowledge-Based Systems*, 89:1 – 13, 2015.

[27] J. Zhang, Y. Wang, C. Zhang, and Y. Shi. Mining contiguous sequential generators in biological sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, PP(99):1–1, 2015.

Appendix

NP-hardness of Lexis Problem

We prove that the Lexis problem is NP-hard through a reduction from the *Smallest Grammar Problem* (SGP) [6].

Formally, *The Smallest Grammar Problem* for a string s is to identify a *Straight-Line Grammar* (SLG) G^* such that $L(G^*) = \{s\}$ and $|G^*| \leq |G|$ for any other G with $L(G) = \{s\}$, where $|G|$ denotes the size of grammar G. Charikar et al. [6] define the size of a grammar as the cumulative length of the right-hand side of all rules, i.e., $|G| = \sum_{T \to \alpha \in \Delta} |\alpha|$ where $|\alpha|$ is the number of symbols appearing in the term α of a grammar rule. Under this grammar size, Charikar et al. show that the Smallest Grammar Problem is NP-hard [6].

THEOREM 1. *The Lexis problem in Eq. (2) is NP-hard for $C(D)$ being edge cost (Eq. (3)) or concatenation cost (Eq. (4)).*

Proof: Let us first start with edge cost. Consider an instance of SGP in which we are given string s and we are asked to compute an SLG G such that $L(G) = \{s\}$ and $|G| \leq m$, where $|G| = \sum_{T \to \alpha \in \Delta} |\alpha|$. We reduce it to an instance of the Lexis problem with a single target string $T = \{s\}$, in which we are asked to compute a Lexis-DAG D with $\mathcal{E}(D) \leq m$.

Given a grammar $G = (\Sigma, \Gamma, S, \Delta)$ as a solution to the SGP problem, we construct a solution D for the reduced Lexis problem. For each symbol in $\Sigma \cup \Gamma$, construct a node. For a non-terminal $T \in \Gamma$, we refer to the corresponding node also as T, and associate that node with the string $\mathcal{S}(T)$ that is produced by expanding rule T according to grammar G. Also, for each rule $T \to \alpha$ in G, we scan α and add an edge in D from every node that corresponds to a terminal or nonterminal in α to the node that corresponds to T (along with the corresponding index). It is easy to see that D is acyclic since G is a straight-line grammar and that the number of edges in D is: $\mathcal{E}(D) = \sum_{T \to \alpha \in \Delta} |\alpha| \leq m$.

Conversely, consider a Lexis-DAG $D = (V_S \cup V_M \cup V_T, E)$ which is a solution to the Lexis problem from our reduction above , i.e., it has a single target string s, and $\mathcal{E}(D) \leq m$. We can construct a corresponding grammar G for the SGP as follows. For each source node in V_S, construct a terminal in Σ. For each node v in $V_T \cup V_M$, construct a nonterminal $NT(v)$ in Γ. For the single target node v in V_T, designate $NT(v)$ as the start symbol S. For each node v in $V_T \cup V_M$, add a rule in Δ with the right-hand side listing the corresponding $\Sigma \cup \Gamma$ symbols for all nodes in the sequence $I(v)$ (i.e., ordered as their respective strings appear concatenated in $\mathcal{S}(v)$). The constructed grammar is straight-line, since every nonterminal has one rule associated with it, and the grammar is also acyclic because the Lexis-DAG D is acyclic. It is easy to see that $|G| \leq m$.

The NP-hardness proof of the Smallest Grammar Problem with grammar size defined as $|G| = \sum_{T \to \alpha \in \Delta} |\alpha|$ [6] can be adapted for a modified grammar size definition, i.e., $|G| = \sum_{T \to \alpha \in \Delta} (|\alpha| - 1) = (\sum_{T \to \alpha \in \Delta} |\alpha|) - |\Delta|$. We can then use the same reduction from SGP to Lexis as in the case of edge cost to show that Lexis with concatenation cost is also NP-hard. ∎

Lexis-DAGs: Edge Cost vs Concatenation Cost

Fig. 6 illustrates an example that the Lexis-DAG when optimizing for the edge cost function may be different than the Lexis-DAG when optimizing for concatenation cost.
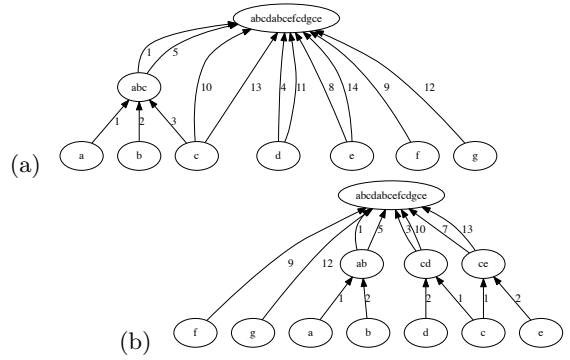

(a)

(b)

Figure 6: Illustration of Lexis-DAGs for target $T = \{abcdabcefcdgce\}$ and sources $S = \{a, b, c, d, e, f, g\}$ - **(a)** Lexis-DAG D_1 with $\mathcal{E}(D_1) = 13$ (optimal) and $\mathcal{C}(D_1) = 11$ (suboptimal). **(b)** Lexis-DAG D_2 with $\mathcal{E}(D_2) = 14$ (suboptimal) and $\mathcal{C}(D_2) = 10$ (optimal).

Comparison of G-LEXIS with Longest Substring Replacement Algorithm

We compare G-LEXIS with an algorithm that greedily replaces the longest repeated substring, in terms of both runtime and cost. We implemented the latter, originally proposed in [13] using suffix trees, using our own efficient linked-suffix array. We used the NSF data described in the main text and ran the two algorithms on different fractions of the total dataset, repeating the experiments 10 times and recording the average runtime and edge cost. As seen in Fig. 7, the Longest Substring Replacement heuristic offers a better runtime but its cost becomes increasingly worse than G-LEXIS as the size of the dataset grows. Also, G-LEXIS is still reasonably fast on all datasets we have analyzed so far.
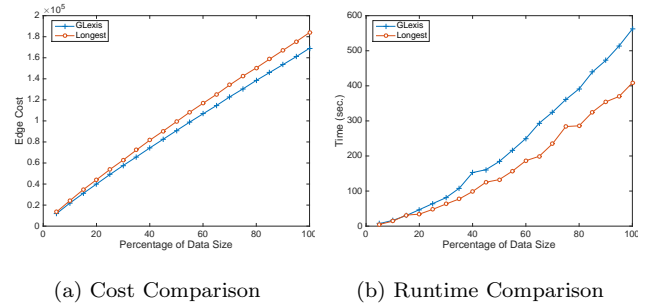

(a) Cost Comparison   (b) Runtime Comparison

Figure 7: Comparison of G-LEXIS with Longest Substring Replacement algorithm.

Central DNA Parts in iGEM Dataset

Table 5: Top-15 nodes with highest path-centrality iGEM dataset's Lexis-DAG. Nodes shown in bold are already registered in the iGEM library.

B0010-B0012	**E0040-B0010-B0012**
B0034-E0040-B0010-B0012	**B0034-C0062-B0010-B0012**
B0032-E0040-B0010-B0012	B0034-C0062-B0010-B0012-R0062
B0034-E1010-B0010-B0012	**B0034-E1010**
B0034-I732006-B0034-E0040-B0010-B0012	**B0034-C0062**
B0030-E0040-B0010-B0012	**R0010-B0034**
K228000-B0010-B0012	**R0040-B0034**
C0051-B0010-B0012	