Constructing an Indeterminate String from its Associated Graph

A Thesis Presented to

The Faculty of the Computer Science Department

California State University Channel Islands

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Joel Helling

Advisor: Michael Soltys

May 2017

© 2017
Joel Helling
ALL RIGHTS RESERVED

Constructing an Indeterminate String from its Associated Graph

Joel Helling

May 16, 2017

Abstract

In this thesis, we first establish a conjecture given in [Christodoulakis *et al.*, **Indeterminate strings, prefix arrays and undirected graphs**, *Theoret. Comput. Sci.* 600–4 (2015)] that the cardinality of a basis $\mathcal{B}$ of the maximal cliques in a finite simple graph $\mathcal{G}$ of order n is exactly the size σ of the minimum alphabet of an indeterminate string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ whose ***associated graph*** $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. We go on to prove that the computation of such a $\mathcal{B}$ for given $\mathcal{G}$ is an NP-complete problem, and then we describe a heuristic algorithm that computes a basis $\mathcal{B}'$, $|\mathcal{B}'| \geq |\mathcal{B}|$, which covers $\mathcal{G}$. We also show that, for any graph $\mathcal{G}$ without 3-cliques, $|\mathcal{B}'| \leq \lfloor n^2/4 \rfloor$; hence that, on such graphs with exactly $\lfloor n^2/4 \rfloor$ cliques, our algorithm is optimal. We continue by showing, using previous results, that $\lfloor n^2/4 \rfloor$ is in fact the upper bound for the size of a basis for *all* graphs.

Contents

List of Figures

1 Introduction

In this thesis, we seek to extend the connections between graph theory and stringology explored in [5]. We consider a ***string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ to be a sequence of ***letters*** $\boldsymbol{x}[i], 1 \leq i \leq n$, that are nonempty subsets of the elements of a given finite set Σ, called the ***alphabet***. If $\boldsymbol{x}[i]$ is a subset of cardinality 1, it is said to be a ***regular*** letter; otherwise, ***indeterminate***. Similarly, if $\boldsymbol{x}$ contains only regular letters, it is said to be ***regular***; otherwise, ***indeterminate***. For example, on $\Sigma = \{a, b, c\}$, $\boldsymbol{x} = ababc$ is regular, while $\boldsymbol{y} = \{a, b\}ba\{b, c\}b$ is indeterminate.

Given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, we say that for $1 \leq i, j \leq n$, $\boldsymbol{x}[i]$ ***matches*** $\boldsymbol{x}[j]$ (written $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$) if and only if $\boldsymbol{x}[i] \cap \boldsymbol{x}[j] \neq \emptyset$. Thus $\boldsymbol{x}[i] = \boldsymbol{x}[j] \implies \boldsymbol{x}[i] \approx \boldsymbol{x}[j]$. As defined in [5], the ***associated graph*** $\mathcal{G}_{\boldsymbol{x}} = (V_{\boldsymbol{x}}, E_{\boldsymbol{x}})$ of $\boldsymbol{x}$ is the simple graph whose vertices are positions $1, 2, \ldots, n$ in $\boldsymbol{x}$ and whose edges are the pairs (i, j) such that $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$. We say that $\boldsymbol{x}$ is ***essentially regular*** whenever $\boldsymbol{x}[i] \approx \boldsymbol{x}[j] \implies \boldsymbol{x}[i] = \boldsymbol{x}[j]$ for every i, j. Hence every essentially regular string can be replaced by an equivalent regular one, and the associated graph of $\boldsymbol{x}$ is a collection of cliques if and only if $\boldsymbol{x}$ is essentially regular.

For indeterminate strings, however, $\mathcal{G}_{\boldsymbol{x}}$ is more interesting. In Section 4 we begin by proving a conjecture stated in [5], that given a finite simple graph $\mathcal{G}$ whose maximal cliques have basis $\mathcal{B}$, then $|\mathcal{B}|$ is the minimum alphabet size of any string $\boldsymbol{x}$ whose associated graph $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. We go on in Section 5 to show that the problem of determining a basis for a given graph $\mathcal{G}$ is NP-complete. Section 6 describes an algorithm that approximates a basis of $\mathcal{G}$ by assigning letters to the vertices of cliques until all vertices are labeled, thus effectively computing a string $\boldsymbol{x}$ for which $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. Experimental results of Algorithm 1 are shown in Section 7. We will begin with Section 3, examining preliminary research which motivates this thesis.

This is an example of the "reverse engineering" of a data structure, a class of problem initiated in [12, 11] for the border array, and extended to other structures in, for example, [2, 13, 6].

2 Contribution

This author was a contributor on [15], on which this thesis is based, along with the other three authors. The paper [15] was published in the Journal

of Theoretical Computer Science.

This author's specific contributions include: the proposal and development of Algorithm 1, the correctness and upper bounds for the number of symbols needed to label a graph as shown in Lemmas 6, 8, 9, and 10. This author also implemented Algorithm 1 in Python to produce Figures 4, 5, and 3 to provide feedback on the general performance of Algorithm 1. Additionally, this author created Figures 2 and 6 to provide examples of the output of Algorithm 1.

3 Background

String algorithms have focused on searching for a pattern, a specified sequence of characters, in a much larger corpus of text. Very efficient pattern search algorithms were presented in [4] and [18] that greatly improve upon the naive approach by pre-processing of the pattern to enhance the speed on the search by allowing the search to skip parts of the text being searched. The naive approach does a character by character comparison of the pattern and searched text which has a worst-case time complexity of $O(nm)$, where n is the length of the pattern and m is the length of the text. The algorithm put forward in [4], known as the Boyer-Moore algorithm, is interesting, because instead of starting the search by comparing the first character in the pattern with the first character in the text, it starts comparisons at the end of the pattern. The preprocessing that is done on the pattern produces multiple tables of information which determines how many characters to the right to shift the pattern in the search string and begin comparisons again to maximize the amount of comparisons that are skipped. This algorithm is able to search for the pattern much more quickly than the naive approach and works best on text that uses a large alphabet (i.e. natural language). The Knuth-Morris-Pratt algorithms [18], conversely, searches the text similarly to the naive algorithm, but uses a single "Partial Match" array to determine how part to shift the pattern to the right for comparisons. This algorithm works very well for strings with very small alphabets such as DNA sequences which only use A, T, G, and C.

The second major area of focus for string algorithms is repeating substrings (of any length), referred to as repetitions. The first paper on Stringology by Axel Thue [27] demonstrated an algorithm to generate an infinite string on an alphabet of only three letters. For repetitions, the most famous

theoretical result in the area of repetitions is the "periodicity lemma":

Lemma 1 *[9] Let p and q be two periods of $\boldsymbol{x}$, and let $d = gcd(p, q)$. If $p + q \leq |x| + d$, then d is also a period of $\boldsymbol{x}$.*

This lemma provides a mechanism to calculate the lengths of periods in strings. Aside from the string search algoritm that is shown in [18], the "Partial Match" array used for the algorithm is equivalent to calculating the period length of each prefix of a string. This is also called the Border Array:

Definition 2 *[5] If a string $\boldsymbol{x}$ can be written $\boldsymbol{x} = \boldsymbol{u_1 v}$ and $\boldsymbol{x} = \boldsymbol{w u_2}$ for nonempty string $\boldsymbol{v}, \boldsymbol{w}$, where $\boldsymbol{u_1} \approx \boldsymbol{u_2}$, then $\boldsymbol{x}$ is said to have a **border** of length $|\boldsymbol{u_1}| = |\boldsymbol{u_2}|$.*

*The **border array** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an integer array $\boldsymbol{\beta}[1..n]$ such that $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$.*

However, the border array specifies the border of every prefix of $\boldsymbol{x}$, but, for indeterminate strings and their intransitive match characteristics, that is not necessarily true. The prefix array has no such deficiencies:

Definition 3 *[5] The **prefix array** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is the integer array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ such that for every $i \in 1..n, \boldsymbol{y}[i]$ is the length of the longest prefix of $\boldsymbol{x}[i..n]$ that matches the prefix of $\boldsymbol{x}$.*

Let $\boldsymbol{w} = aa\{a, b\}b$, then the prefix array $\boldsymbol{y} = 4310$. The prefix array also specifies all of the borders of every prefix, which the border array cannot for indeterminate strings [25].

Indeterminate strings were first introduced in [10] as using a "Don't Care" symbol in the alphabet of both the pattern and the search text as a generalization of the algorithm presented in [18]. One of the main areas of research for indeterminate strings is pattern matching which is motivated by its applications to the field of computation biology introduced further below. Pattern matching for indeterminate strings adds complications to the existing algorithms, because the definition of ***match*** is expanded. The authors of [26] have created a novel approach to pattern matching by combining other algorithms to remove the dependence on the border array which is no longer accurate for indeterminate strings. Other authors have also put forward efficient algorithms for indeterminate strings, see [22, 23].

Indeterminate strings have been useful in various application areas, including bioinformatics, crytpanalysis, search engines, etc. In computational biology, DNA sequences can be encoded as long text strings, moreover finding a pattern in the string allows for the letters A/T to be swapped with C/G, repectively, and still be considered a match for the pattern. Pattern matching is used in bioinformatics to identify proteins and DNA sequences with the concept of alignment. Alignment is matching the pattern sequence of DNA with a sequence that exists in the database. Two major software suites are used to query sequence databases, BLAST [1] and FASTA [21], which are used in a significant amount of computational biology research. An example in bioinformation can be found in [7], where Doerr et al. created an efficient computational model to predict the occurence of gene clusters in chromosomal genetic data by looking at the common intervals of the indeterminate strings. The common interval of indeterminate strings, s and t, is when $C(s[i..j]) = C(t[p..q])$ and C is the set of characters in the substring of the indeterminate string. Using such data, Doerr et al. was able to infer the family of the gene clusters of the chromosomal DNA, which was required as input in previous work. Another example of indeterminate strings in biology is from [17]. Pattern matching on indeterminate patterns and text with multiple patterns, and looking for the maximal overlap in matching regions. The algorithms can also be applied to computer vision to combine multiple images of the same object.

Indeterminate strings can be used in cryptanalysis. With more users, it is inceasingly necessary to utilize cloud infrastructure to host users data. For their data to be secure, however, the data must be encrypted. The use of encrypted searches can be used to ensure that users will be able to securely be able to access their data, which will never need to be decrypted for the search. A Cyphertext-Policy Hidden Vector Encryption scheme allows from encrypted searches while still allowing only policy based access to certain parts on the text [3]. The scheme detailed in [28] allowed wildcards to create policies for users. These policies allow the search of encrypted data by multiple users by selecting wildcards when choosing which user is allowed to access a particular search.An example would be a two departments, one with access and one without access, trying to search an encrypted database. The department without access would not be able to search any of the data, while the department with access would be able to send search queries even through all of the members of that deparment have different keys.

Indeterminate strings can also play a role in traffic phasing problems as

described in [24]. The traffic phasing problem takes streams of traffic and schedules times that each stream is able to access some resource. Streams that are compatible are able to overlap. The problem can be represented by a graph $\mathcal{G}$ where each vertex is a stream, and there is an edge between vertices that have compatible time periods, $S(\boldsymbol{x})$, where $S(\boldsymbol{x})$ is the set of time periods that a stream needs at a resource. We can the define compatability as $S(\boldsymbol{x}) \cap S(\boldsymbol{y}) \neq 0$ [24]. This problem may be able to be mapped onto indeterminate strings with the goal being a reduction in the number of symbols that would be used to label the associated graph.

The previous research that inspired this thesis was [5], which showed a link between prefix arrays, indeterminate strings, and undirected graphs. We provide Algorithm 1 as a solution to the "reverse engineering" problem of converting from an undirected graph to an indeterminate string, and provide put forward properties of those indeterminate strings pertaining particularly to the number of symbols they use.

4 Maximal Cliques in the Associated Graph $\mathcal{G}_{\boldsymbol{x}}$

Let $\mathcal{G}$ be a finite simple graph. Recall that a ***clique*** C is a complete subgraph of $\mathcal{G}$; C is said to be ***maximal*** if it is not a subgraph of any other clique. Let $\mathcal{S}$ be the set of all maximal cliques in $\mathcal{G}$, and let $\mathcal{B}$ be a smallest subset of $\mathcal{S}$ such that every vertex and every edge of $\mathcal{G}$ occur in some member of $\mathcal{B}$. Then $\mathcal{B}$ is said to be a ***basis*** of the maximal cliques of $\mathcal{G}$. Note that there may be more than one basis for a given $\mathcal{G}$ (for an example, see Figure-1).

Lemma 4 *Suppose that a finite simple graph $\mathcal{G}$ with vertex set $V = \{1, 2, \ldots, n\}$ has a basis $\mathcal{B}$ of maximal cliques of cardinality σ. Then there is a string $\boldsymbol{x}$ on a base alphabet of size σ whose associated graph $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. No string on a smaller alphabet has this property.*

Proof. Let $\mathcal{B} = \{C_1, C_2, \ldots, C_\sigma\}$. Let $\{\lambda_s\}_{s=1}^{\sigma}$ be distinct letters. We construct a string $\boldsymbol{x}$ as follows. For each ordered pair (s, i) with $1 \leq s \leq \sigma$ and $1 \leq i \leq n$, assign λ_s to $\boldsymbol{x}[i]$ if vertex i occurs in the maximal clique C_s. It is clear from the definitions that the string $\boldsymbol{x}$ so constructed satisfies $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$.

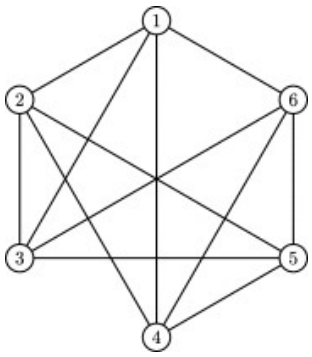


Figure 1: Graph on six vertices that can be partitioned into four maximal cliques in two ways: {123, 146, 245, 356} or {456, 124, 235, 136} [5].

Now consider any string $\boldsymbol{x}$ of length n for which $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$ and let τ be the number of distinct (ordinary) letters occurring in $\boldsymbol{x}$. For each such letter λ, there is a clique C_λ of $\mathcal{G}$ whose vertices are those i for which $\lambda \in \boldsymbol{x}[i]$. Of course, these cliques may not be maximal, but each C_λ can be extended to a maximal clique C'_λ. Note that every vertex and edge of $\mathcal{G}$ occurs in one of the cliques C_λ and *a fortiori* in one of the maximal cliques C'_λ. However, the C'_λ might not all be distinct. Let τ' be the number of distinct C'_λ. Then $\tau \geq \tau' \geq \sigma$, the latter inequality following from the fact that there is a basis of cardinality σ. This shows that τ cannot be less than σ and completes the proof. □

5 Computing Maximal Cliques is NPC

As mentioned at the end of Section 1, we are interested in "reverse engineering" a data structure, which in our case means that for a given $\mathcal{G}$ we want to find a string $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$, where we may impose additional conditions on $\mathcal{G}$ and $\boldsymbol{x}$. At an abstract level, the problem may be stated as follows: given an "empty" data structure that satisfies certain properties, can we populate it with data that satisfies those properties? It is easy to see that this is the problem of satisfiability in disguise, and so the NPC results of this section are to be expected. Here we show that, given a graph $\mathcal{G}$ and a parameter K, the problem of finding $\boldsymbol{x}$ over an alphabet Σ, $|\Sigma| \leq K$, such

that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$, is NPC.

Problem [GT17] in [14, p. 194], COVERING BY CLIQUES, is defined as follows:

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |E|$.

QUESTION: Are there $k \leq K$ subsets $V_1, V_2, \ldots, V_k$ of V such that each V_i induces a complete subgraph (clique) of G and such that for each edge $(u, v) \in E$ there is some V_i that contains both u and v?

For a given graph $\mathcal{G}$ we are interested in computing σ, the minimum number of labels necessary to label $\mathcal{G}$ in such a way that two nodes share a label if and only if they share an edge. By Lemma 4 we know that σ is the size of any basis $\mathcal{B}$ of $\mathcal{G}$. Since we know that $|\mathcal{B}| \leq |E| \leq |V|^2 = n^2$, we can reduce the computation of σ to COVERING BY CLIQUES in polynomial time by doing a binary search for the smallest K for which the answer to the question is "yes." This smallest K is our σ. On the other hand, COVERING BY CLIQUES can also be reduced in polynomial time to our problem of computing σ as follows: if $K \geq \sigma$, then the answer to COVERING BY CLIQUES is clearly "yes," as $k = |\mathcal{B}| = \sigma \leq K$; while if $K < \sigma$, then the answer is "no." To see this, note that if $V_1, V_2, \ldots, V_k$, $k \leq K$ were a covering by (not necessarily maximal) cliques, then each V_i could be extended to some maximal V_i', and so, eliminating possible duplicates, $V_{j_1}', V_{j_2}', \ldots, V_{j_{k'}}'$, where $k' \leq k \leq K < \sigma$, would be a basis, giving us a contradiction.

Thus computing the size of $\mathcal{B}$ for a given graph $\mathcal{G}$ is NPC, which by Lemma 4 means that given a graph $\mathcal{G}$, computing the size of the smallest alphabet for an indeterminate $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$ is also NPC. In other words:

Lemma 5 *Given a graph $\mathcal{G}$, finding an indeterminate $\boldsymbol{x}$ on a minimum alphabet such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$ is an NPC problem.*

6 Graph Labeling Algorithm

In this section we introduce an algorithm which takes as input a graph $\mathcal{G}$, and outputs a labeling that respects its edge relations, thus effectively an $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. Our algorithm works by exploring maximal cliques, and assigning all the vertices in a given clique the same symbol; as in general

vertices may be located in several cliques, they may get several labels. We want the labeling to be as frugal as possible, but since in the last section we showed that the problem of finding a string with minimum alphabet size is NPC, we cannot expect a polynomial time procedure, such as Algorithm 1, to furnish us with an optimal labeling. Nevertheless, Algorithm 1 is simple, and performs well, as we discuss in this section. Essentially, it inspects all vertices w adjacent to each vertex v; then, by considering all other vertices q also adjacent to v, it seeks to find a largest clique containing the edge (v, w).

Algorithm 1 Given the adjacency lists of the n vertices of $\mathcal{G}$, compute a set of cliques that cover $\mathcal{G}$ so that each vertex i has a set of labels (letters) specifying a corresponding string entry $\boldsymbol{x}[i]$.

```
1:  procedure LabelGraph(v.adj:Adjaceny List for v, v.label:{} ∀ v)
2:      λ ← 1
3:      for v ← 1 to n do
4:          if v.degree = 0 then
5:              v.label ← {λ}
6:              λ ← λ + 1
7:          else
8:              for all w ∈ v.adj do
9:                  if v.label ∩ w.label = ∅ then
10:                     v.label ← v.label ∪ {λ}
11:                     w.label ← w.label ∪ {λ}
12:                     clique ← {w}
13:                     for all q ∈ v.adj − {w} do
14:                         if clique ⊆ q.adj then
15:                             q.label ← q.label ∪ {λ}
16:                             clique ← clique ∪ {q}
17:                     λ ← λ + 1
```

Lemma 6 *Algorithm 1 is correct; i.e., given G as input, it outputs $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$.*

Proof. First note that λ is updated on every pass in which a label is assigned. Thus every label assigned determines a unique v, namely the pass in which that label is assigned. In particular, each vertex of degree 0 is assigned its

own unique label. We now show that for vertices of positive degree,

$$\boldsymbol{x}[v] \approx \boldsymbol{x}[w] \iff (v,w) \in E \quad (\text{where } \mathcal{G} = (V,E)). \tag{1}$$

Suppose that $(v,w) \in E$. In step v of the outer for-loop on line 3, we will eventually consider w as $w \in v.adj$ on line 8. If $v.label \cap w.label \neq \emptyset$ on line 9, then $\boldsymbol{x}[v] \approx \boldsymbol{x}[w]$. Otherwise, we assign λ to both (line 10,11), and reach the same conclusion.

Conversely, suppose that $\boldsymbol{x}[w_1] \approx \boldsymbol{x}[w_2]$. Then there is a label λ that the algorithm assigns to w_1 and w_2. These assignments take place on a specific pass, say v. If $v \notin \{w_1, w_2\}$, we must have (in order for any assignments to occur), $w \in v.adj$ with $v.label \cap w.label = \emptyset$. Then λ is assigned to v and w and *clique* is set equal to w. If $w \notin \{w_1, w_2\}$, then the assignments of λ to w_1 and w_2 must take place in line 15. This requires that one of them (say w_1) is adjacent to w and then w_2 is adjacent to both w and w_1. In particular, $(w_1, w_2) \in E$. On the other hand, if $w = w_1$ (or equivalently, w_2), then w_2 must be assigned the label λ in line 15, which requires that w_2 is adjacent to w_1, the only element of *clique* at this point.

Now consider the possibility that $v \in \{w_1, w_2\}$. Without loss of generality, $v = w_1$. If $w_2 \in v.adj$, then $(w_1, w_2) \in E$. Otherwise, there exists $w \in v.adj$ and both v and w are assigned the label λ in lines 10 and 11 respectively. In order for w_2 to be assigned the label λ in line 15, we need w_2 to be adjacent to $v = w_1$.

Thus, we may conclude $\boldsymbol{x}[w_1] \approx \boldsymbol{x}[w_2]$ implies $(w_1, w_2) \in E$. □

Note that the worst-case running time of Algorithm 1 is $O(n^4)$, due to the three nested **for** loops at lines 3, 8, 13 and the check on line 14 that *clique* is a subset of $q.adj$ which takes a linear amount of time. Also note that we assume that the adjacency relation is given as a set of lists: each vertex v has an associated adjacency list $v.adj$ that lists all the other vertices that are connected to it by an edge. A lot depends on how these lists are populated; i.e., how $\mathcal{G}$ is implemented. Suppose that $\mathcal{G}$ is presented as follows: for each clique C_i in a basis $\mathcal{B}$, we select a representative vertex v_i, and reorder the vertices in V so that $v_1, v_2, \ldots, v_\sigma$ are processed first, therefore considered first in the loop on line 3. Then by listing all the w's in C_i first in the adjacency list of v_i — that is, in $v_i.adj$ — we will have refined the algorithm to run in such a way that it traces the cliques in $\mathcal{B}$ and returns an optimal labeling.

Lemma 7 *The upper bound on the size of the alphabet produced by Algorithm 1 is $n(n-1)/2$, where n is the number of vertices in $\mathcal{G}$.*

Proof. Consider a naive algorithm that works as follows. First it assigns a new symbol λ_v to each isolated vertex v. The remaining vertices are all adjacent on some edge; so it considers all the edges $e = (u, v)$ in $\mathcal{G}$, and assigns a new symbol λ_e to u and to v. This creates an correct labeling, though one that is not optimal in general. Still this procedure shows us that the optimal labeling, whatever it is, is always bounded above by the maximum possible number of edges in an undirected graph, which is $\binom{n}{2} = n(n-1)/2$. Further, it can be easily seen that the naive algorithm is not optimal. Consider a triangle, i.e., a clique on three edges. The optimal solution employs one symbol; the naive algorithm described above uses three. Incidentally, Algorithm 1 also produces an optimal solution in this case.

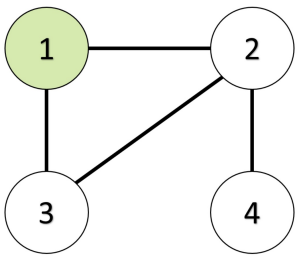
Now Algorithm 1 does no worse than this naive procedure. It assigns labels to all the isolated vertices in lines 4–6 in a way that is optimal. Then the two loops (starting at line 3 and starting at line 8) consider all the edges, and they assign at most one new label per edge, as can be seen from line 9. Algorithm 1 may in fact perform better than the naive procedure as all the vertices in a given clique may be dispatched with the same label. See Figure 2. □

Lemma 8 *Given a graph $\mathcal{G}$ that does not contain any 3-cliques, Algorithm 1 will produce a labeling with alphabet size at most $\lfloor n^2/4 \rfloor$.*

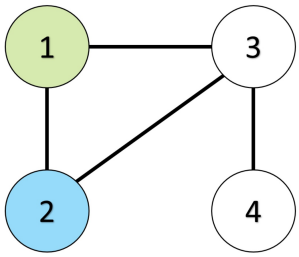
Proof. Ignore singletons which contribute one symbol each. Otherwise, we will always skip lines 15 and 16 in Algorithm 1 because there are no 3-cliques, and the condition in line 14 is never satisfied. Thus we effectively add one symbol per edge on line 10. By Mantel's Theorem [20], a graph without 3-cliques can have at most $\lfloor n^2/4 \rfloor$ edges, and so at most as many labels. □

Lemma 9 *Given a graph $\mathcal{G}$ on $\lfloor n^2/4 \rfloor$ edges that does not contain any 3-cliques, a correct labeling requires an alphabet size of at least $\lfloor n^2/4 \rfloor$. Thus Algorithm 1 is optimal on such graphs.*

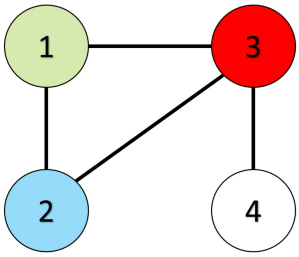
Proof. By way of contradiction, suppose that fewer than $\lfloor n^2/4 \rfloor$ symbols are required for a correct labeling. From Lemma 4 we know that if $\mathcal{B}$ is a basis,



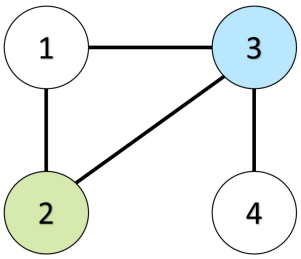
(a) Vertex 1 is selected as the first v.



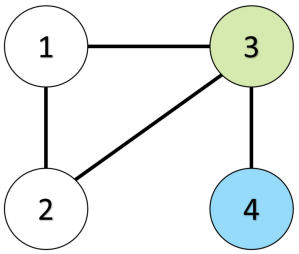
(b) Vertex 2 is then selected as w. The test on line 9 passes, and $\lambda = 1$ is added to both 1 and 2.



(c) Vertex 3 is selected as q. Since 3 is adjacent to both 1 and 2, the test on line 13 passes and $\lambda = 1$ is added to 3.



(d) Since vertices 1,2, and 3 all have the same label ($\lambda = 1$), the check on line 9 fails $w = 3$ and all of the vertices adjacent to $v = 2$.



(e) Vertex 3 is now v, and the checks on line 9 fail for $w = 1, 2$. The test passes on $w = 4$, so $\lambda = 2$ is added to 3 and 4. Vertex 4 as v fails all checks on line 9, and Algorithm 1 ends.

Figure 2: Example of Algorithm 1 where green is v, blue is w, and red is q. The resulting indeterminate string $\boldsymbol{x} = 11\{1, 2\}2$.

then $|\mathcal{B}| < \lfloor n^2/4 \rfloor$. By the Pigeonhole Principle, this means that there must exist some (maximal) clique C in $\mathcal{B}$ that covers at least two edges. But in turn, this means that C contains at least three vertices, and hence a 3-clique, giving us a contradiction. Therefore, $\mathcal{G}$ requires at least $\lfloor n^2/4 \rfloor$ symbols. □

Lemma 10 *Given a graph $\mathcal{G}$, the maximum number of labels needed for a correct labeling is at most $\lfloor n^2/4 \rfloor$.*

Proof. As shown in [8] and restated in [24], let $\theta_e(\mathcal{G})$ be the size of the smallest set of cliques that covers all of the edges of $\mathcal{G}$. Then, $\theta_e(\mathcal{G})$ of any $\mathcal{G}$ without isolated vertices is at most $\lfloor n^2/4 \rfloor$. Since $\theta_e(\mathcal{G})$ is the size of the smallest such covering, $\theta_e(\mathcal{G}) = |\mathcal{B}|$. Therefore, $|\mathcal{B}| \leq \lfloor n^2/4 \rfloor$. Thus, the number of labels required for a correct labeling $\leq \lfloor n^2/4 \rfloor$. To account for the isolated vertices c, we can remove them from the graph to get $\mathcal{G}'$ which has $n - c$ vertices. This gives a $|\mathcal{B}'| \leq \lfloor (n-c)^2/4 \rfloor$, and $|\mathcal{B}| = |\mathcal{B}'| + c < \lfloor n^2/4 \rfloor$ for $c \geq 1$. □

From Lemma 10, we have an upper bound on the number of labels for any graph. However, as shown in [19], this bound can be reduced with the number of edges in the graph. Futher, the case where $|E| > \lfloor n^2/4 \rfloor$ is of interest, because a graph with fewer edges can be covered by a $\mathcal{B}$ such that $|\mathcal{B}| \leq |E|$. From the result of [19], let $k = \binom{n}{2} - |E|$, and t be the greatest natural number such that $t^2 - t \leq k$. Then $\mathcal{G}$ can be covered by $|\mathcal{B}| \leq k + t$.

The problem COVERING BY CLIQUES discussed in Section 5 was shown in [14, p. 193] to be NPC by a reduction from PARTITION INTO CLIQUES [GT15], defined as follows:

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |E|$.

QUESTION: Can the vertices of G be partitioned into $k \leq K$ disjoint sets $V_1, V_2, \ldots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a clique?

Note that a partition into cliques is not necessarily a covering by cliques, because a partition does not require that for every edge (u, v) there be a clique containing both u and v. In light of this, recall from Section 1 that $\boldsymbol{x}[i]$ matches $\boldsymbol{x}[j]$ if and only if they share a common symbol. It is easy to

see that $\approx$ is, as a relation, both reflexive and symmetric, but not necessarily transitive. However, it is clear that given a graph $\mathcal{G}$, we can find indeterminate $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$, where $\boldsymbol{x}$ is on an alphabet of size K, and $\approx$ on $\boldsymbol{x}$ is transitive (hence an equivalence relation) provided $(\mathcal{G}, K)$ is a "yes" instance of PARTITION INTO CLIQUES, where the partition is *also* a covering. But in that case, $\boldsymbol{x}$ is simply a regular string, and so $\approx$ is trivially transitive. Thus we have the following interesting consequence of Algorithm 1:

Lemma 11 *Given a graph $\mathcal{G}$, whether there exists* regular $\boldsymbol{x}$ *such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$ can be determined in polynomial time using Algorithm 1.*

Proof. The algorithm examines all cliques one by one, tracing each out until complete, and assigning a single symbol to it. This is repeated for each clique, and yields a single label for each clique. □

Thus, if $\mathcal{G}$ can be represented with a regular $\boldsymbol{x}$, Algorithm 1 will establish that in polynomial time. But, if $\mathcal{G}$ cannot be represented with a regular $\boldsymbol{x}$, Algorithm 1 will find in polynomial time an indeterminate $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$, but the alphabet of this indeterminate $\boldsymbol{x}$ will not be necessarily minimal.

7 Implemenation of Algorithm-1

Implentation of Algorithm-1 was done with the Python programming language (Python Sofware Foundation, https://www.python.org/) version 3.4.4 with the matplotlib library[16] to graph results. The source code is available at: https://github.com/joelhelling/GraphIndeterminates/tree/master/Python.

Initially the implementation of Algorithm-1 was a nearly a direct translation from the previous psuedocode with the use of Python's standard Set data structure. This implementation performed correctly but was unable to deal with graphs that were larger than 100 vertices. The bulk of computation time was spent performing generic Set operations which run in linear time.

The speed of the Set operations was improved by implementing the Set operations as Bit manipulations, where each intersection and subset can be checked with a logical And. The original Adjacency List and list of labels was replaced with Python's built in integer type used as a bit vector.This change provided a major increase in the number of vertices that were able

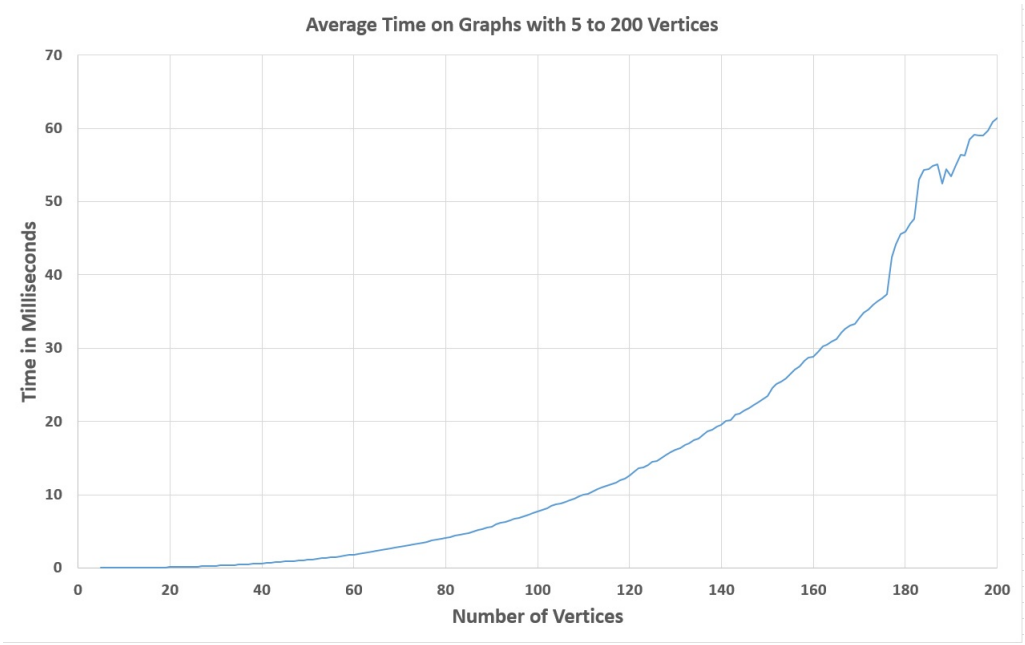


Figure 3: This is a plot of the average time used to label a graph. The x-axis is the number of vertices in the graph, and the y-axis is the average time used to label 100 random graphs on $n = 5, \ldots, 200$ vertices.

to be in the graph and have processing complete. However, a limitation of this method is an increase in the amount of memory used for the resulting labels and adjacency list due to the use of expanding integers in the Python runtime, which can cause out of memory errors for graphs that are too large (i.e. more than 1000 vertices).

The average running time used to label graphs on $n = 5, \ldots, 200$ vertices is shown in Figure 3. The graph shows that the average time to label each individual graph increases exponentially with the size of the graph.

8 Discussion of Results

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two undirected simple graphs or order n whose vertices are labeled $1, 2, \ldots, n$. $\mathcal{G}_1$ and $\mathcal{G}_2$ are said to be ***distinct*** if and only if there exists no permutation of the vertex labels of $\mathcal{G}_2$ such that the adjacency sets

of $\mathcal{G}_1$ and $\mathcal{G}_2$ are pairwise identical. Let $\mathcal{G}^{(n)}$ denote the set of all distinct graphs of order n.

For any graph $\mathcal{G} \in \mathcal{G}^{(n)}$, let $\mathcal{B}$ denote a basis of the set of cliques (that is, by Lemma 4, a minimum alphabet of the associated string), and let $b = |\mathcal{B}|$. Denote by $b^{(n)}$ the average value of b over all $\mathcal{G} \in G^{(n)}$.

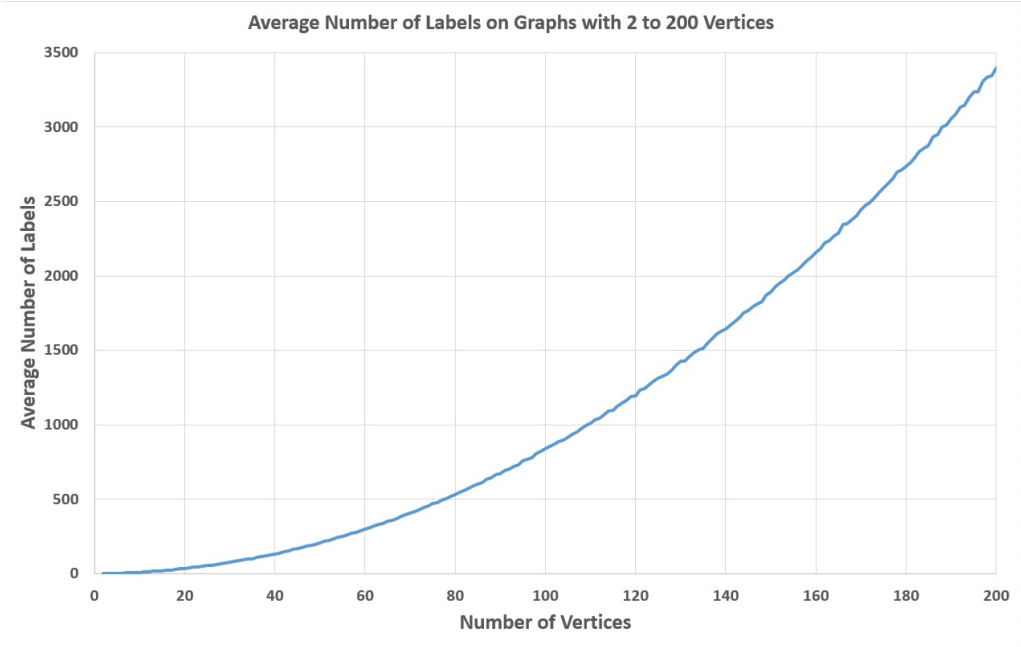


Figure 4: This is a plot of the average number of symbols that are used to label a graph. The x-axis is the number of vertices in the graph, and the y-axis is the average number of symbols used to label 100 random graphs on $n = 2, \ldots, 200$ vertices.

Now consider the process of computing a graph $\mathcal{G}'$ in $\mathcal{G}^{(n+1)}$ from a graph $\mathcal{G}$ in $\mathcal{G}^{(n)}$. All the graphs of $\mathcal{G}^{(n+1)}$ can be formed by adding a single vertex v with label $n+1$, then adding $t = 0, 1, \ldots, n$ edges in all possible ways to the vertices of each $\mathcal{G} \in \mathcal{G}^{(n)}$. The graphs formed in this manner will however not be distinct. For $t = 0$, a single new graph will be introduced, whose basis $b_{n+1} = b_n + 1$, where b_n is the basis of $\mathcal{G}$. For $t \geq 1$, every collection of new edges from v that include *every* vertex in a maximal clique of $\mathcal{G}^{(n)}$ will extend the maximal clique by a single vertex, and may, but need not add a single

element to the basis of $\mathcal{G}^{(n+1)}$ that has no equivalent in $\mathcal{G}^{(n)}$. Otherwise, collections of new edges that access only some vertices in a maximal clique of $\mathcal{G}$ will surely add at least one new element.

With this construction in mind, it appears plausible that, especially when graphs are constrained to be distinct, $b^{(n+1)}$ will not be much larger than $b^{(n)}$. As an example, $b^{(1)} = 1$, $b^{(2)} = 1.5$, $b^{(3)} = 2$ and $b^{(4)} = 29/11$. This idea is reinforced by Figure 4, which shows the growth of Algorithm 1's output as n ranges from 5 to 200, and leads to the following

Conjecture 12 $b^{(n)} \in O(n \log n)$.

9 Conclusion and future work

From the analysis of Algorithm 1, we see that the problem is easy when there are very few edges (e.g., isolated vertices require only as many labels as vertices), also when there are a lot of vertices (e.g., a clique requires just one label). So the problem is difficult somewhere in between. As shown in Figure 5, this intuition is borne out by experimental data.

We have found the x-axis point where graphs such as those of Figure 5 reach the maximum. We show in Lemma 10 that it is $\lfloor n^2/4 \rfloor$. We also think that the graph might be a "saw-tooth" graph, because every graph on n vertices is the subgraph of some graph on $n + 1$ vertices, so that maxima from smaller graphs may be reflected as local maxima in the bigger graph.

We also conjecture that our algorithm is optimal up to a relabeling of the vertices. That is, for every graph $\mathcal{G}$, there exists a graph $\mathcal{G}'$ isomorphic to $\mathcal{G}$ such that the alphabet of the indeterminate string $\boldsymbol{x}$ produced by Algorithm 1 applied to G' has size σ. We discussed this briefly following the proof of Lemma 6, where we point out that Algorithm 1 will return a minimal labeling as long as we can "rig" the order of the vertices in the outer loop, and the ordering of each adjacency list to effectively trace out the cliques in a given basis $\mathcal{B}$ (this of course requires knowing the basis $\mathcal{B}$ before hand). Here we conjecture that there is one universal ordering of the vertices, so that the vertices in each adjacency list are also listed in that order, so that given that ordering of the graph, Algorithm 1 will return an optimal $\boldsymbol{x}$. See Figure 6.

We can also look to fast heuristics that can improve upon the resulting labeling without increasing the worst-case running time of Algorithm 1. An example of such a heuristic would be sorting the vertices based on the degree

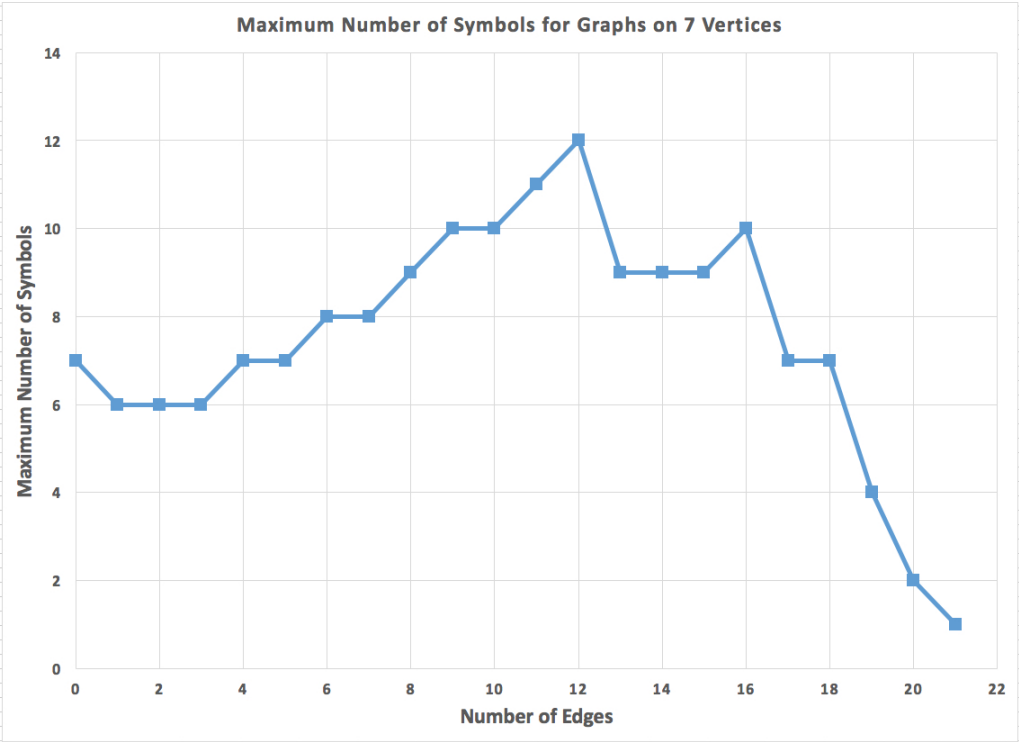


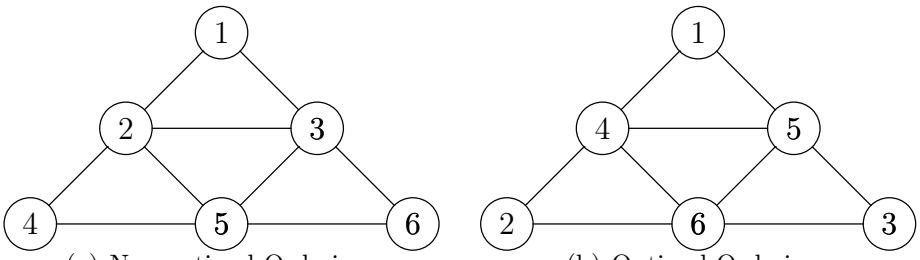
Figure 5: This is a plot of all graphs on 7 vertices, where the x-axis is the number of edges (from 0 to a max of 21), and the y-axis is the maximum number of labels over all the graphs with a fixed number of edges. Note the low number of labels at the extremes, and the spike in the middle.

of the vertex. Another more complex heuristic can be sorting the Adjacency List of $v(v.adj)$ with respect to the number of labels that are currently assigned to each vertex in the Adjacency List.

We finally present open questions related to graph labeling and the Algorithm 1:

1. What are efficient heurstics that can be applied to Algorithm 1 to produce labeling results that are closer to optimal?

2. What is the upper-bound on the number of labels that Algorithms 1 produces over the minimal result?



(a) Non-optimal Ordering (b) Optimal Ordering

Figure 6: Algorithm 1 on (a) will produce $\boldsymbol{x} = 1\{1,2,3\}\{1,3,4\}2\{2,3,4\}4$, while (b) will produce $\boldsymbol{x} = 123\{1,2\}\{1,3\}\{2,3\}$. This results in a smaller alphabet for (b), which in this case is minimal.

3. In light of Lemma 11, is there a faster way (say, $O(n^2)$) algorithms to determine whether, given $\mathcal{G}$, there exists a regular $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$?

4. What stuctures of a graph can be inferred from its associated indeterminate string?

References

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[2] Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In Branislav Rovan and Peter Vojt's, editors, *Symp. on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003.

[3] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. *Public Key Encryption with Keyword Search*, pages 506–522. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[4] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[5] Manolis Christodoulakis, P. J. Ryan, W. F. Smyth, and Shu Wang. Indeterminate strings, prefix arrays and undirected graphs. *Theoretical Comput. Sci.*, 600(4):34–48, 2015.

[6] Julien Clément, Maxime Crochemore, and Giuseppina Rindone. Reverse engineering prefix tables. In *Proc. 26th Symp. on Theoretical Aspects of Computer Science*, pages 289–300, 2009.

[7] Daniel Doerr, Jens Stoye, Sebastian Böcker, and Katharina Jahn. Identifying gene clusters by discovering common intervals in indeterminate strings. *BMC Genomics*, 15:S2, 2014.

[8] Paul Erdös, A.W. Goodman, and Louis Pósa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18:106–112, 1966.

[9] N.J. Fine and H.S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[10] M.J. Fischer and M.S. Paterson. String matching and other products. In R.M. Karp, editor, *Complexity of Computation,*, pages 113–125. American Mathematical Society, 1974.

[11] Frantisek Franek, Shudi Gao, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear time. *J. Combinatorial Maths. and Combinatorial Comput.*, 42:223–236, 2002.

[12] Frantisek Franek, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear timee (preliminary version). *Proc. 10th Australasian Workshop on Combinatorial Algs. (AWOCA) School of Computing, Curtin University of Technology*, pages 26–33, 1999.

[13] Frantisek Franek and W. F. Smyth. Reconstructing a suffix array. *Internat. J. Foundations of Computer Science*, 17(6):1281–1296, 2006.

[14] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP–Completeness*. W. H. Freeman, 1979.

[15] Joel Helling, P.J. Ryan, W. F. Smyth, and Michael Soltys. Constructing an indeterminate string from its associated graph. *Theoretical Computer Science*, 2017.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science And Engineering*, 9(3):90–95, 2007.

[17] Costas S. Iliopoulos, Ritu Kundu, and Manal Mohamed. *Efficient Computation of Clustered-Clumps in Degenerate Strings*, pages 510–519. Springer International Publishing, Cham, 2016.

[18] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[19] László Lovász. On covering graphs. Proceedings of the Colloquium, pages 231–236, Tihany, Hungary, 1968. Academic Press, New York.

[20] W. Mantel. Problem 28 (solution by H. Gouweniak, W. Mantel, J. Texeira de Mattes, F. Schuh, and W.A Whythoff). *Wiskundige Opgaven*, 10(60–61), 1907.

[21] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448.

[22] Ron Y. Pinter. *Efficient String Matching with Don't-Care Patterns*, pages 11–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[23] M. Rahman and C. Iliopoulos. Pattern matching algorithms with dont́ cares. *SOFSEM*, 2:116–126, 2007.

[24] Fred S. Roberts. Applications of edge coverings by cliques. *Discrete Applied Mathematics*, 10:93–109, 1985.

[25] W. F. Smyth and Shu Wang. New perspectives on the prefix array. *Proc. 15th SPIRE, Lecture Notes in Computer Science, LNCS 5280, Springer Verlag*, 27:133–143, 2008.

[26] W. F. Smyth and Shu Wang. An adaptive hybrid pattern-matching algorithm on indeterminate strings. *Internat. J. Foundations of Computer Science*, 20(6):985–1004, 2009.

[27] Axel Thue. über unendliche zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (7):1–22, 1906.

[28] Liao Zhenhua, Wang Jinmiao, and Lang Bo. A ciphertext-policy hidden vector encryption scheme supporting multiuser keyword search. *Security Comm. Networks*, 8:879–887, 2015.