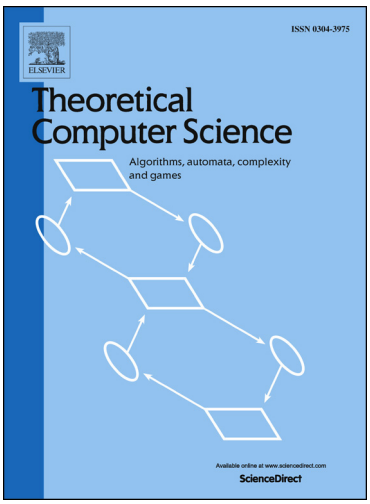


Constructing an indeterminate string from its associated graph

Joel Helling, P.J. Ryan, W.F. Smyth, Michael Soltys

PII: S0304-3975(17)30149-4
DOI: http://dx.doi.org/10.1016/j.tcs.2017.02.016
Reference: TCS 11085

To appear in: *Theoretical Computer Science*

Received date: 21 May 2016
Revised date: 2 February 2017
Accepted date: 14 February 2017

Please cite this article in press as: J. Helling et al., Constructing an indeterminate string from its associated graph, *Theoret. Comput. Sci.* (2017), http://dx.doi.org/10.1016/j.tcs.2017.02.016

Highlights

- We propose a new algorithm for the problem of reverse engineering an indeterminate string from an arbitrary undirected graph.
- Our algorithm is simple and faster than other algorithms, though it yields larger alphabets.
- We clarify the interconnections between clique covers, intersection numbers and alphabet sizes for indeterminate strings.
- We have created a software suite for running experiments, and we present several experimental results in the paper.

Constructing an Indeterminate String from its Associated Graph*

Joel Helling[1], P. J. Ryan[2], W. F. Smyth**[2,3], and Michael Soltys[1]

[1] Department of Computer Science
California State University Channel Islands
{joel.helling904,michael.soltys}@csuci.edu
[2] Algorithms Research Group
Dept. of Computing and Software, McMaster University
{ryanpj,smyth}@mcmaster.ca
[3] School of Engineering and Information Technology
Murdoch University

Abstract. As discussed at length in [Christodoulakis *et al.*, **Indeterminate strings, prefix arrays and undirected graphs**, *Theoret. Comput. Sci.* 600–4 (2015)], there is a natural one-many correspondence between simple undirected graphs $\mathcal{G}$ with vertex set $V = \{1, 2, \ldots, n\}$ and ***indeterminate strings*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ — that is, sequences of subsets of some alphabet Σ. In this paper, given $\mathcal{G}$, we consider the "reverse engineering" problem of computing a corresponding $\boldsymbol{x}$ on an alphabet $\Sigma_{\min}$ of minimum cardinality. This turns out to be equivalent to the NP-hard problem of computing the ***intersection number*** of $\mathcal{G}$, thus in turn equivalent to the ***clique cover*** problem. We describe a heuristic algorithm that computes an approximation to $\Sigma_{\min}$ and a corresponding $\boldsymbol{x}$. We give various properties of our algorithm, including some experimental evidence that on average it requires $\mathcal{O}(n^2 \log n)$ time. We compare it with other heuristics, and state some conjectures and open problems.

Keywords: String Algorithms; Indeterminate Strings; Cliques; Graph Labeling

1 Introduction

In this paper we seek to extend the connections between graph theory and stringology explored in [3]. We consider a ***string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ to be a sequence of ***letters*** $\boldsymbol{x}[i], 1 \le i \le n$, that are nonempty subsets of a given finite set Σ, called the ***alphabet***. If $\boldsymbol{x}[i]$ is a subset of

* We are grateful to Manolis Christodoulakis and Shu Wang for helpful discussions.
** Supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

cardinality 1, it is said to be ***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[4], while $\boldsymbol{y} = \{a, b\}ba\{b, c\}b$ is indeterminate. Indeterminate strings are useful in various application areas, notably bioinformatics, where under certain circumstances DNA sequences can be regarded as indeterminate strings on nucleotides $\{a, c, g, t\}$. In recent years indeterminate strings have been the subject of much study [12, 11, 17, 1].

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 in particular $\boldsymbol{x}[i] = \boldsymbol{x}[j] \Longrightarrow \boldsymbol{x}[i] \approx \boldsymbol{x}[j]$. As defined in [3], 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 $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$. Suppose that for some position $i_0 \in 1..n$, $\boldsymbol{x}[i_0]$ matches $\boldsymbol{x}[i_1], \boldsymbol{x}[i_2], \ldots, \boldsymbol{x}[i_k]$ for some $k \geq 0$, and matches no other elements of $\boldsymbol{x}$. We say that position i_0 is ***essentially regular*** if and only if the entries in positions $i_1, i_2, \ldots, i_k$ match each other pairwise. If every position in $\boldsymbol{x}$ is essentially regular, we say that $\boldsymbol{x}$ itself is ***essentially regular***. Hence every essentially regular string can be replaced by an equivalent regular one, and the associated graph $\mathcal{G}_{\boldsymbol{x}}$ is a collection of disjoint cliques if and only if $\boldsymbol{x}$ is essentially regular.

For general indeterminate strings, however, $\mathcal{G}_{\boldsymbol{x}}$ is more interesting. In Section 2 we discuss a conjecture stated in [3], that given a finite simple graph $\mathcal{G}$ whose maximal cliques have basis $\mathcal{B}$, $|\mathcal{B}|$ is the minimum alphabet size of any string $\boldsymbol{x}$ whose associated graph $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. We discover that this conjecture is just a reformulation, in a slightly different context, of the problem of computing the intersection number of $\mathcal{G}$, which is NP-hard, and hence computing the minimum alphabet size of any string is also NP-hard. Section 3 describes an algorithm that approximates a basis of $\mathcal{G}$ by assigning symbols to the vertices of cliques until all vertices are labeled, thus effectively computing a string $\boldsymbol{x}$ whose associated graph $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. This is an example of the "reverse engineering" of a data structure, a class of problems initiated in [8, 7] for the border array, and ex-

[4] For singleton sets such as $\{a\}, \{b\}, \{c\}$, we write a, b, c for simplicity.

tended to other structures in, for example, [2, 9, 4]. In Section 4 we discuss our algorithm's results and execution, especially in the context of other algorithms that perform closely-related computations. Section 5 discusses a few conjectures and open problems.

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

Suppose a collection $\mathcal{F} = F_1, F_2, \ldots, F_n$ of sets is given. Then the ***intersection graph*** $\mathcal{G}_{\mathcal{F}}$ of $\mathcal{F}$ is a simple undirected graph on $|\mathcal{F}| = n$ vertices $1, 2, \ldots, n$, with an edge (i, j), $1 \leq i, j \leq n$, if and only if $F_i \cap F_j \neq \emptyset$. Conversely, it was shown in [18] that, given a simple undirected graph $\mathcal{G}$ on vertices $1, 2, \ldots, n$, a collection $\mathcal{F}$ of n sets can be found such that $\mathcal{G}$ is the intersection graph of $\mathcal{F}$. (For example, for each (i, j) in $\mathcal{G}$, $i < j$, place a unique symbol $\lambda_{i,j}$ in F_i and F_j.) The ***intersection number*** $\theta(\mathcal{G})$ of $\mathcal{G}$ is the smallest number of distinct symbols that can be placed in the sets of $\mathcal{F}$ such that $\mathcal{G} = \mathcal{G}_{\mathcal{F}}$. In our application, the collection $\mathcal{F}$ becomes a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ with $\boldsymbol{x}[i] = F_i$ (necessarily nonempty). The associated graph and the intersection graph are thus the same, and we seek a smallest alphabet $\Sigma_{\min}$ that produces it. Let $\sigma_{\min} = |\Sigma_{\min}|$, the cardinality of such an alphabet.

The standard way of efficiently representing a finite simple graph $\mathcal{G}$ as an intersection graph is by covering the graph by cliques. Take any set of cliques covering all edges of $\mathcal{G}$. For each vertex v, let F_v be the set of those cliques containing the vertex v. Then the intersection graph of $\{F_v\}$ coincides with $\mathcal{G}$. As a result, the intersection number $\theta(\mathcal{G})$ is equal to the ***edge clique cover number*** $ec(\mathcal{G})$, the cardinality of a minimum size set of the cliques that covers all the edges of $\mathcal{G}$. Erdős *et al.* [6, 16] proved that $\theta(\mathcal{G}) \leq \lfloor n^2/4 \rfloor$, an upper bound that is achieved when $\mathcal{G}$ is a triangle-free graph on $\lfloor n^2/4 \rfloor$ edges [15]. An instructive example is given by the complete bipartite graphs $K_{m,m}$ (for even $n = 2m$) and $K_{m,m+1}$ (for odd $n = 2m + 1$) for which the minimal covering by cliques consists of all edges, the number of which is precisely $\lfloor n^2/4 \rfloor$.

Erdős *et al.* use coverings that cover all vertices as well as all edges. If $\mathcal{G}$ has no isolated points, this is equivalent to the "edge covering" approach discussed above. For this case, they prove that

$ec(\mathcal{G}) \leq \lfloor n^2/4 \rfloor$ and that one need only use 2-cliques and 3-cliques (edges and triangles) in a minimal covering.[5]

More recently, Conjecture 21 in [3] formulated the problem in a slightly different way, in terms of the ***maximal*** cliques in $\mathcal{G}$; that is, those that are not proper subgraphs of any other clique. We provide here a proof of this conjecture, thus validating the several following remarks made in that paper. To be consistent with [3], we use the notion of ***basis***. A basis is a minimum size set of maximal cliques that covers all edges and all vertices of $\mathcal{G}$.

Lemma 1 *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 $\sigma_{\min}$. Then there is a string $\boldsymbol{x}$ on a base alphabet of size $\sigma_{\min}$ 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 $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$.

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 $\sigma_{\min}$. This shows that τ cannot be less than $\sigma_{\min}$ and completes the proof. □

It turns out that maximality is irrelevant in the specification of basis. Let $\phi'(\mathcal{G})$ be the cardinality of a basis (of maximal cliques) in

[5] The authors thank an anonymous referee who drew their attention to the available material on intersection graphs and clique edge covers.

$\mathcal{G}$ and let $\phi(\mathcal{G})$ be the cardinality of a smallest set of cliques that cover all edges and vertices of $\mathcal{G}$. Then:

Observation 2 $\phi'(\mathcal{G}) = \phi(\mathcal{G})$.

Proof. If $\phi'(\mathcal{G}) < \phi(\mathcal{G})$, then $\phi(\mathcal{G})$ cannot be minimum, a contradiction. Suppose then that $\phi(\mathcal{G}) < \phi'(\mathcal{G})$. This requires that the minimum covering ϕ includes cliques that are not maximal. But these cliques can be extended to become maximal cliques, so that therefore there is minimum covering by maximal cliques that is less than ϕ', again a contradiction. □

For graphs with no isolated vertices, every edge-covering by cliques also covers all vertices. When there are isolated vertices, we need an additional clique for each. Thus we have the following:

Observation 3 *Let $\mathcal{G}$ be a finite simple graph with d vertices of degree 0. Then*

$$\sigma_{\min} = \phi(\mathcal{G}) = ec(\mathcal{G}) + d.$$

It is well known [14, 10] that computing $\theta(\mathcal{G})$ is NP-hard, thus so also is the computation of $\sigma_{\min} = \theta(\mathcal{G}) + d$. In the next section we describe a heuristic algorithm to compute an approximation of $\sigma_{\min}$.

3 Graph Labeling Algorithm

Our algorithm accepts as input a graph $\mathcal{G}$, and outputs a labeling that respects its edge relations, thus effectively producing an $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$. The 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 the computation is NP-hard, we cannot of course expect a minimum labelling. Nevertheless, Algorithm 1 is simple, and performs well in practice, as we discuss below. Essentially, it runs three nested loops: all vertices v, then all vertices w adjacent to v, and finally all vertices q adjacent to both v and w and previous q's. For these latter

vertices q we apply a crucial heuristic that reduces dramatically the number of labels required for most graphs: we consider them in non-decreasing order $l(q)$ of already-assigned labels, as discussed below. For an example of Algorithm 1's execution, see Figure 1.

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 (symbols) specifying a corresponding string entry $\boldsymbol{x}[i]$.

```
 1: procedure LabelGraph(v.adj,n:v.label)
 2:     λ ← 1
 3:     for v ← 1 to n do
 4:         if v.deg = 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
```

First we examine the running time of Algorithm 1, showing that on the surface the run time is $O(n^6)$, but with some simple tweaks this can be improved and brought down to $O(n^4)$. We have nested loops in lines 3,8,13, together with implied loops in lines 9,14, where each line costs $O(n)$, except line 9 where we have to check if among all labels (at most $O(n^2)$ of them) v, w share one. Thus the cost of line 9 is $O(n^2)$. Line 14 contributes in a similar manner, in that we have to make sure that all the vertices in the clique (at most $O(n)$ of them) are in the adjacency list of vertex q. Further, as noted above, line 13 is executed in $l(q)$ order; that is, in non-decreasing order of number

of labels already assigned to the vertices q. Thus $v.adj - \{w\}$ needs to be sorted before the execution of line 13, a process that requires $O(n \log n)$ steps in the worst case, less than that required by lines 13 and 14, thus a negligible time penalty. The worst-case bound therefore remains at $O(n^6)$, but in practice the algorithm runs much faster (in Section 4 we we conjecture in $O(n^2 logn)$ steps on average over constant graph density).

Run time is also affected by other design decisions related to data structures and optimizations used. On lines 10, 11 and 15, instead of adding labels to a resulting set of labels for each vertex, vertex-label pairs (v, λ) could have instead been stored in a list, then at the end of the v-loop radix-sorted and merged into the final output. This would have avoided the difficulty resulting from the fact that a single vertex could be an entry in as many as n–1 cliques. Instead we used dynamically expanding arrays to keep track of labels, a decision that, except for pathological cases, turns out to have little impact on processing time.

However, we can certainly improve the upper bound to $O(n^4)$ by counting more carefully and by modifying the algorithm slightly, to amortize the cost of line 9. The point to note is that to compute 9 we do not need to find the particular label that v, w share in common, but only check whether such a label exists.

Instead we check whether v, w share a label by maintaining a 0-1 matrix M which has a 1 in position (v, w) if and only if $v.label \cap w.label \neq \emptyset$ (and so in line 9 of the algorithm we only check if $M_{vw} \neq 1$, which can be done in one step). Of course, there will be an added cost of maintaining the matrix M, an overall cost of $O(n^4)$, which can be amortized in the total runtime of the algorithm. In order to maintain M, we add a couple of lines after line 17, with the same indentation as line 17, which set $M_{vw} = 1, M_{vq} = 1, M_{wq} = 1$, and all the corresponding entries reflected along the main diagonal (as the graph is undirected), for all w, q in the clique where $w \neq q$. This takes at most $O(n^2)$ steps.

Now, each entry (u, v) in the matrix M may stay zero throughout (if u, v are not adjacent), or be changed to 1 once or repeatedly. Note that each entry that is turned to 1 will be turned to 1 at most $O(n^2)$ times as that is the maximum number of cliques produced by the algorithm (according to our Remark 6), and any entry (u, v) will be

turned to 1 as many times as the number of different cliques that contain edge (u, v).

This means that updating matrix M cannot cost more than $O(n^4)$, the desired running time of the entire algorithm. This total can be amortized over the entire run, so that the maintenance of matrix M does not increase the running time of the algorithm, while it allows line 9 to be counted as a single step. Hence:

Lemma 4 *The worst-case running time of Algorithm 1 is $O(n^4)$.*

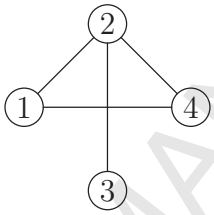


Fig. 1. Algorithm 1 begins with $\lambda \leftarrow 1$ and will first select v_1 as v on line 3. Since $v.deg \neq 0$, v_2 will be selected as the first w, so that the test in line 9 passes, λ is added to $v.label$ and $w.label$, and *clique* replaced by w. Line 13 will select v_4 as q, and the test on line 14 will pass, adding λ to $q.label$ and q to *clique*. The loop on line 13 ends and λ is incremented. v_4 is the next w, but the check on line 9 will now fail, so v_2 is the next v. The test on line 9 will fail with $w = v_1$, but pass with $w = v_3$. $\lambda = 2$ is added to $v.label$ and $w.label$. v_1 and v_4 as q will fail the test on line 14. v_3 and v_4 as v will fail the rest of the checks on line 9. The resulting $\boldsymbol{x} = 1\{1, 2\}21$, best possible.

Lemma 5 *Algorithm 1 is correct; that is, given $\mathcal{G}$ as input, it outputs $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$.*

Proof. We first observe that all labels assigned in lines 10 through 15 are to vertices of positive degree and that λ is updated if such an assignment takes place during a pass. On the other hand, λ is also updated whenever line 5 is executed. Thus each vertex of degree 0 is assigned its own unique label and so it does not match any other

vertex. 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, since the latter is an 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 find $\boldsymbol{x}[w_1] \approx \boldsymbol{x}[w_2]$ implies $(w_1, w_2) \in E$. □

We conclude this section with some remarks on the effectiveness and efficiency of Algorithm 1.

Remark 6 *The upper bound on the size of the alphabet produced by Algorithm 1 is m, the number of edges in $\mathcal{G}$, plus the number of isolated vertices.*

Remark 7 *Algorithm 1 is optimal on triangle-free graphs, in particular on maximal triangle-free graphs with $\lfloor n^2/4 \rfloor$ edges.*

Given a graph $\mathcal{G}$ on n vertices and m edges, a straightforward greedy algorithm determines in $\Theta(m)$ time whether or not $\mathcal{G}$ is a union of cliques, merely by inspecting the adjacencies of each vertex

not included in a previously inspected clique. Since Algorithm 1 also adopts a greedy approach to the adjacencies of each vertex v considered, we have:

Remark 8 *Algorithm 1 is optimal for all graphs $\mathcal{G}$ that are a disjoint union of cliques; that is, it assigns to each vertex a single letter defining a regular string $\boldsymbol{x}$ such that $\mathcal{G}_{\boldsymbol{x}} = \mathcal{G}$.*

4 Discussion of Algorithmic Results

In this section we compare the performance of Algorithm 1 against two other algorithms:

- an edge clique covering algorithm proposed in the 1970s by Kellerman [13];
- a recently proposed algorithm due to Conte *et al.* [5].

The number of cliques computed by both of these algorithms is reduced by post-processing proposed by Kou *et al.* [14].

Note that each of these algorithms produces the resulting edge clique covering while Algorithm 1 produces sets of labels. As shown in Lemma 1, we can consider the outputs of the algrithms as equivalent, and translating between the two can also be done in linear time. The test suite containing the benchmarks is written in the Java programming language version 1.8.0_111.

Tables 1-3 show the number of labels required and corresponding timings for tests of the three algorithms on randomly-generated graphs with edge densities 0.1, 0.5 and 0.8, respectively. Algorithm 1 is generally faster than CGM, especially for more dense graphs, while the Kellerman algorithm is an order of magnitude slower than either of the other two. As discussed in Section 5 (Remark 13), this advantage for Algorithm 1 can be extended, especially for denser graphs, by parallel execution.

In terms of labels, however, CGM yields somewhat better results on sparser graphs, but holds a wide advantage over both Kellerman and Algorithm 1 for densities 0.5 and 0.8. The CGM result for density 0.5 is particularly surprising, since it drops substantially from the CGM result for density 0.1: as discussed in Section 5 (see Figure 3), the sizes of minimum clique covers are likely to be greatest at the

middle of the density range. It is puzzling that CGM, designed for sparse graphs, should actually execute much better on those that are not sparse.

100 Random Graphs: $n = 1000,\ m = 50,000$		
Algorithm	**Average Number of Labels**	**Average Run Time (seconds)**
Kellerman w/ Kou	17,400	31.6
CGM w/ Kou	15,700	0.2
Algorithm 1	19,000	0.2

Table 1. Graphs on 1000 vertices with edge density 0.1.

100 Random Graphs: $n = 1000,\ m = 250,000$		
Algorithm	**Average Number of Labels**	**Average Run Time (seconds)**
Kellerman w/ Kou	19,100	74.5
CGM w/ Kou	11,500	2.0
Algorithm 1	21,600	1.4

Table 2. Graphs on 1000 vertices with edge density 0.5.

In order to estimate the average case run time of Algorithm 1, we executed it on 100 randomly-generated graphs of orders $n = 500, 600, \ldots, 1500$ for each of the three edge densities $0.1, 0.5, 0.8$. Over each set of 100 runs, the average time required per run was

100 Random Graphs: $n = 1000,\ m = 400,000$		
Algorithm	**Average Number of Labels**	**Average Run Time (seconds)**
Kellerman w/ Kou	9,200	29.5
CGM w/ Kou	3,700	2.8
Algorithm 1	8,200	0.8

Table 3. Graphs on 1000 vertices with edge density 0.8.

computed. For densities 0.1 and 0.8, the first differences of the timings appeared to be linear in n, slightly concave upward for 0.5.

The interested reader may download the software from the Github repository. The implementation of Algorithm 1, the implementations of the other two algorithms in the above tables (Kellerman and CGM), as well as the text files containing the results of running the experiments, are contained in the directory `Java/LabelAlgorithm` in: `https://github.com/joelhelling/GraphIndeterminates`

5 Conjectures and Open Problems

Based on the experiments described in Section 4, we first state the following:

Conjecture 9 *In the average case, for fixed edge density, Algorithm 1 executes in* $\mathcal{O}(n^2 \log n)$ *time.*

As remarked earlier, Algorithm 1 inspects the q's in line 13 in non-decreasing order of $l(q)$, where $l(q)$ is the number of labels already assigned to q. Without this heuristic, Algorithm 1 would perform much worse in practice: in Tables 1, 2, 3, the Average Number of Labels would be 28,700, 37,000, 29,900, respectively. The reason seems to be, intuitively, that with the heuristic we cover the graph with fewer cliques, as we go first for those vertices that are in no cliques or in fewer cliques.

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two undirected simple graphs of order n each with vertices $1, 2, \ldots, n$. $\mathcal{G}_1$ and $\mathcal{G}_2$ are said to be ***distinct*** if and only if there exists no permutation of the vertices of $\mathcal{G}_2$ such that the adjacency (edge) sets of $\mathcal{G}_1$ and $\mathcal{G}_2$ are pairwise identical; that is, they are distinct if they are not isomorphic. 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 1, a minimum alphabet of the associated string), and let $b = |\mathcal{B}|$. Denote by $b^{(n)}$ the average value of b over all $\mathcal{G} \in \mathcal{G}^{(n)}$.

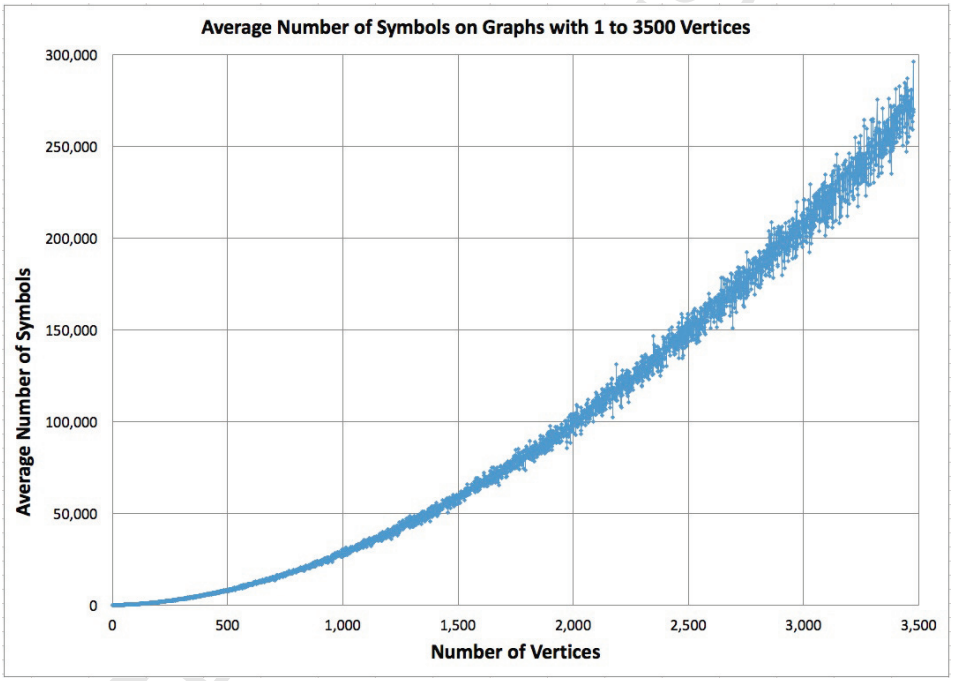


Fig. 2. This is a plot of the average number of symbols that are used to label vertices of graphs. The x-axis is the number of vertices in the graph, and the y-axis is the average number of symbols used to label the vertices of 100 random graphs on $n = 1, \ldots, 3500$ vertices.

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, with $\phi(\mathcal{G}') = \phi(\mathcal{G}) + 1$, where the 1 accounts for the isolated singleton. For $t \geq 1$, every collection of new edges from v that include *every* vertex in a maximal clique of $\mathcal{G}$ will extend the maximal clique by a single vertex, and may or may not add a single element to the basis of $\mathcal{G}'$ that has no equivalent in $\mathcal{G}$. 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 2, which shows the growth of Algorithm 1's output as n ranges from 1 to 3,500, and leads to the following:

Conjecture 10 $b^{(n)} \in O(n \log n)$.

From the analysis of Algorithm 1, we see that computing the intersection number $\theta(\mathcal{G})$ is easy when there are very few edges (e.g., isolated vertices and edges that are maximal cliques are easily handled), also when there are many edges (e.g., a complete graph requires just one label). So the problem is difficult somewhere in between. As shown in Figure 3, this intuition is borne out by experimental data.

It is clear from the results mentioned in Section 2 that the graph in Figure 3, as well as all such graphs for any number of vertices n, reach the maximum at $\lfloor n^2/4 \rfloor$ (in Figure 3 where $n = 7$, $\lfloor 7^2/4 \rfloor = 12$, which is indeed the maximum).

We conjecture that such graphs have a "saw-tooth" shape, 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.

Problem 11 *For graphs $\mathcal{G} = \mathcal{G}(n, m)$, i.e., graphs on n vertices and m edges, characterize the shape of the Edge Graph that shows $\theta(\mathcal{G})$ for fixed n and each $m = 0, 1, \ldots, \binom{n}{2}$.*

We also conjecture that our algorithm is optimal up to a reordering of vertices. What we mean by this is that for every graph $\mathcal{G}$,

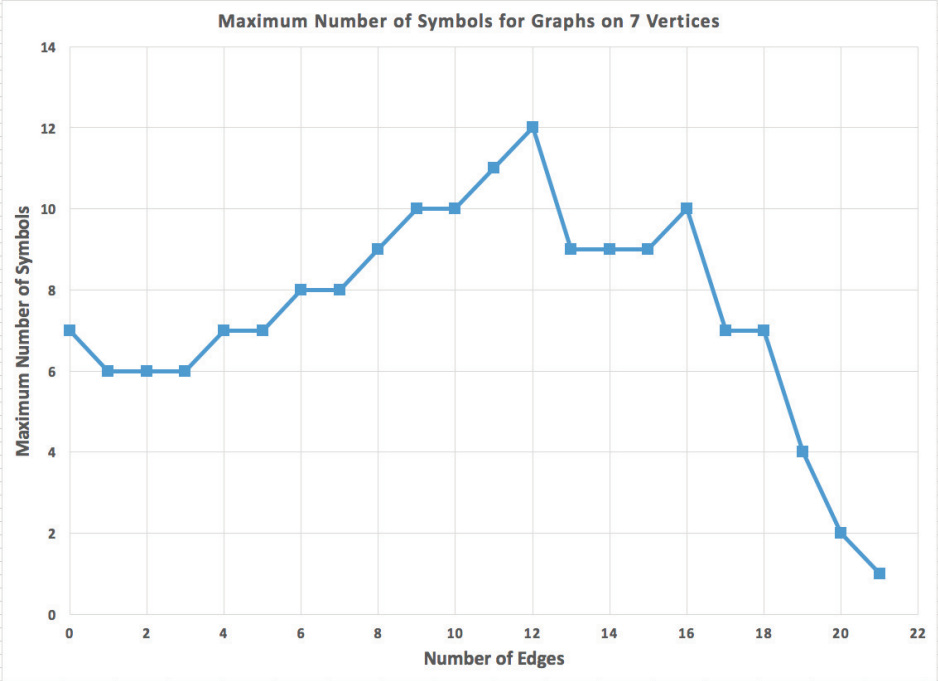


Fig. 3. Edge Graph: this is a plot for all graphs on seven vertices, where the x-axis is the number of edges (0–21), and the y-axis is the maximum number of labels assigned (by an optimal labelling algorithm) over all the graphs with a fixed number of edges. Note the low number of labels at the extremes, the plateaus, and the spikes in the middle.

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 $\mathcal{G}'$ has size $\sigma_{\min}$. In other words, if we can "rig" the order of the vertices in the outer loop and the ordering of each adjacency list, so as to effectively trace out the cliques in a given basis $\mathcal{B}$ (this of course requires knowing the basis $\mathcal{B}$ beforehand), then we get an optimal labeling. Thus a potential strategy to reduce the labels computed by Algorithm 1 is to find a basis for effective reordering of the vertices.

Conjecture 12 *There exists an ordering of the vertices, respected also in the processing of the adjacency lists for each vertex, such that*

Algorithm 1 will return $\boldsymbol{x}$ *on an alphabet of minimum size* $\sigma_{\min} = \phi(\mathcal{G})$. *See Figure 4.*

The natural way to approach Conjecture 12 is to strive to show that there exists an ordering of the vertices that corresponds to the ordering induced somehow by an optimal labelling. Given such a labeling $\{\lambda_1, \lambda_2, \ldots, \lambda_s\}$, a candidate for a good corresponding ordering is:

$$S = C_{\lambda_1}, C_{\lambda_2} - C_{\lambda_1}, \ldots, C_{\lambda_{i+1}} - \bigcup_{j=1}^{i} C_{\lambda_j}, \ldots, C_{\lambda_s} - \bigcup_{j=1}^{s-1} C_{\lambda_j}, \qquad (2)$$

where C_λ is the set of vertices labeled with λ. The idea in (2) is that we order the vertices by labels, making sure that vertices are not repeated. Also note that given any set of vertices in (2), such as C_{λ_1} (vertices labeled with λ_1), or $C_{\lambda_2} - C_{\lambda_1}$ (vertices labeled with λ_2 but not with λ_1), etc., the vertices within any such set can be ordered in an arbitrary manner.

Given an S as in (2), note that the same ordering is employed in line 3 of Algorithm 1 (the main loop), and in lines 8 and 13, where the adjacency list of v, denoted $v.adj$ is examined in the order given by S. Consider what would happen if we were to run Algorithm 1 on the graph given in Figure 4(a). Suppose that $C_{\lambda_1} = \{1, 2, 3\}$, $C_{\lambda_2} = \{2, 4, 5\}$, and $C_{\lambda_3} = \{3, 5, 6\}$, so that $S = 1, 2, 3, 4, 5, 6$. Then Algorithm 1 would assign λ_1 to $v = 1$, and then also λ_1 to $w = 2$, and then it examines all the q's in $1.adj$ which are not w, in the order given in S, so it will find $q = 3$, and assign it λ_1 as well and create C_{λ_1}.

Then, the algorithm will create $C_{\lambda_2} = \{2, 4, 5\}$, but now we have a problem as we examine $v = 3$. The first w in the order given by S with which v does not share a label is $w = 5$, and so 3 and 5 will share λ_3. But now we examine the q's in the order given by S, 2 is connected to both 3 and 5, and so 2 will get labeled with λ_3 as well! Finally, the algorithm will create $C_{\lambda_4} = \{3, 5, 6\}$, producing a non-optimal labeling.

This problem could be avoided by keeping S, but rearranging the adjacency lists to start by exploring the intended clique: in the running example, $3.adj = 5, 6, 1, 2$. In general, the ordering given by

S, and then each $v.adj$ given by first listing the C_λ such that λ is a label of v in the optimal ordering, would work. Still, we conjecture that a canonical ordering, such as the one given in Figure 4(b), exists for every graph. This is Conjecture 12.

Remark 13 *Algorithm 1 can be executed in parallel by partitioning the vertices such that each thread will compute the labeling of the subgraph* $\mathcal{G}_v = \{v \cup v.adj\}$*, where each* v *is not adjacent to any previously selected* v*.*

Since Algorithm 1 only considers v and $v.adj$, we can compute the labeling of each subgraph in a separate thread to parallelize the computation. This can lead to a duplication of cliques when processing two subgraphs $\mathcal{G}_v$ and $\mathcal{G}_{v'}$, but only when $v.adj \cap v'.adj$ contains cliques. There are two ways in which this duplication can be eliminated. First, through the matrix M as described in Section 3 which will prevent duplicate computation of a clique. Second, since the subgraphs are processed in parallel, they may be producing the same clique at the same time. This can be detected by keeping track of the produced cliques and removing cliques that contain the same vertices.

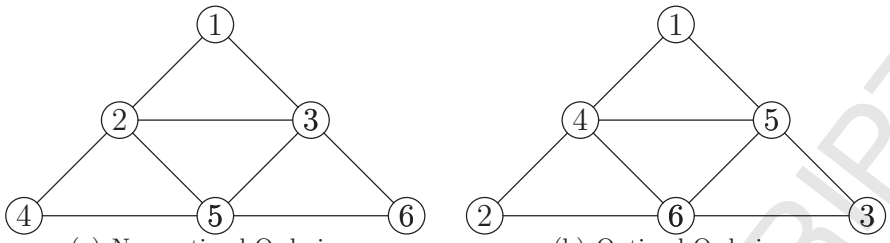
Another consideration is the $l(q)$ heuristic that is updated for every label used, and is used to order the q's before they are processed in line 13. One way of dealing with it is by having global access to $l(q)$, but updating and sorting will most likely create lock contention issues. A different solution would be to have each thread keep track of labels added when processing the current subgraph, but this localized solution may not produce results as favorable as the global approach.

The parallelization of Algorithm 1 could produce considerable speed up on very large sparse graphs, but for dense graphs, the algorithm will converge to the single threaded case.

Appendix: The Erdős-Goodman-Posá proof

Theorem 2 in [6] is stated as follows:

Theorem Any graph $\mathcal{G}^{(n)}$ of order $n \geq 2$ with no isolated vertices can be covered by at most $\lfloor n^2/4 \rfloor$ complete graphs. Further, in the covering we need to use only edges and triangles.



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

Fig. 4. Alg. 1 yields $\boldsymbol{x} = 1\{1,2,3\}\{1,3,4\}2\{2,3,4\}4$ and $\boldsymbol{x} = 123\{1,2\}\{1,3\}\{2,3\}$ on (a) and (b), respectively. This results in a smaller alphabet for (b), which in this case is minimum.

The proof has a gap, but it can be repaired. The problem is that the authors do not deal correctly with subgraphs that have isolated vertices.

First, they observe that the theorem holds for $n = 2$ and $n = 3$. They propose to use induction, reducing a graph $\mathcal{G}^{(n+2)}$ to the $\mathcal{G}^{(n)}$ case. They note that

$$\lfloor (n+2)^2/4 \rfloor = \lfloor n^2/4 \rfloor + n + 1.$$

Now take any edge (x_1, x_2) of a graph $\mathcal{G}^{(n+2)}$ and consider the graph $\mathcal{G}^{(n)}$ obtained by eliminating x_1 and x_2. Thus we have a graph on n vertices and whatever edges join them. They then invoke the induction hypothesis to get a cover for the subgraph, then add back in the $n+1$ or fewer edges or triangles required to cover $\mathcal{G}^{(n+2)}$ to get their result.

Unfortunately, this induction does not apply in general since $\mathcal{G}^{(n)}$ may have isolated vertices. In fact, all vertices might be isolated. Suppose that $\mathcal{G}^{(n)}$ has k non-isolated vertices where $2 \leq k \leq n$. Then the induction hypothesis does apply to this k-vertex subgraph. For the remaining $n-k$ vertices, we can add a triangle or edge for each, depending on whether it is adjacent to both x_1 and x_2 in $\mathcal{G}^{(n+2)}$, or only adjacent to one of them. If the latter holds for every isolated vertex, we need to also add the edge (x_1, x_2). In any case, we add at most $n-k+1$ new complete graphs. Then it is a matter of checking whether

$$\frac{k^2}{4} + n - k + 1 \leq \frac{(n+2)^2}{4}$$

which it is, and then drawing the appropriate conclusions about the integer parts of each. Since we have assumed that $k \geq 2$, we also must consider the possibility that $k = 0$, i.e. all vertices of $\mathcal{G}^{(n)}$ are isolated. In this case the inequalilty becomes

$$4(n+1) \leq (n+2)^2$$

which reduces to $n^2 \geq 0$. Finally, note that $k = 1$ (i.e. $\mathcal{G}^{(n)}$ has $n-1$ isolated vertices) is not possible.

The theorem is also true for $n \geq 4$ if we omit the hypothesis about isolated vertices, provided that we allow 1-cliques as well as 2-cliques and 3-cliques in our cover. To see this, suppose that $\mathcal{G}^{(n)}$ has $n - k > 0$ isolated vertices. We need a 1-clique for each isolated vertex in addition to the cover guaranteed by Theorem 2. To verify the theorem, we need to check that

$$\frac{k^2}{4} + n - k \leq \frac{n^2}{4}$$

which reduces to $n + k \geq 4$ and therefore establishes the theorem for $n \geq 5$. We can check $n = 4$ directly for various k to complete the proof.

Acknowledgement

We thank two referees for their careful reading and useful suggestions that have greatly improved the quality of this paper. We would also like to thank Alessio Conte, Roberto Grossi, and Andrea Marino for providing the source code for their algorithm [5].

References

1. Ali Alatabbi, M. Sohel Rahman, and W. F. Smyth. Inferring an indeterminate string from a prefix graph. *J. Discrete Algorithms*, 32:6–13, 2015.
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. 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.

4. 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.
5. Alessio Conte, Roberto Grossi, and Andrea Marino. Clique covering of large real-world networks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1134–1139, New York, NY, USA, 2016. ACM.
6. Paul Erdös, A. W. Goodman, and Louis Pósa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18:106–112, 1966.
7. 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.
8. 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.
9. Frantisek Franek and W. F. Smyth. Reconstructing a suffix array. *Internat. J. Foundations of Computer Science*, 17(6):1281–1296, 2006.
10. M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP–Completeness*. W. H. Freeman, 1979.
11. J. Holub, W. F. Smyth, and S. Wang. Fast pattern–matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.
12. Jan Holub, W. F. Smyth, and Shu Wang. Hybrid pattern-matching algorithms on indeterminate strings. In Jacqueline W. Daykin, M. Mohamed, and K. Steinhofel, editors, *London Algorithmics and Stringology*, King's College Texts in Algorithmics, pages 115–133. King's College Publications, 2006.
13. E. Kellerman. Determination of keyword conflict. *IBM Technical Disclosure Bulletin*, pages 544–546, 1973.
14. L. T. Kou, L. J. Stockmeyer, and C. K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Comm. ACM*, pages 135–139, 1978.
15. 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.
16. Fred S. Roberts. Applications of edge coverings by cliques. *Discrete Applied Mathematics*, 10:93–109, 1985.
17. 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.
18. Edward Szpilrajn-Marczewski. Sur deux propriétés des classes d'ensembles. *Fundamenta Mathematicae*, 33:303–307, 1945.