The Feasibility and Use of a Minor Containment Algorithm

LIU XIONG and MICHAEL J. DINNEEN
(lxiong@xtra.co.nz & mjd@cs.auckland.ac.nz)

Dept. of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand

Febuary 16, 2000

Abstract

We present a general algorithm for checking whether one graph is a minor of another. Although this algorithm is not polynomial-time, it is quite practical for small graphs. For all connected graphs with 5 vertices or less we count how many connected graphs of order at most 9 are above them in the minor order. Our computed tables may be useful in the design of heuristic algorithms for minor closed families of graphs.

1 Introduction

We are primarily interested in the minor order of graphs for two reasons. First, many families of graphs can be characterized by a finite set of forbidden minors (i.e., structural properties that succinctly express what the members can not possess). Second, the minor relationship between graphs provides us a natural graph (combinatorial) embedding problem.

The general minor containment problem (explained below) to decide for two input graphs G and H, if H is a minor of G is known to be an $\mathcal{NP}$-complete problem [GJ79]. Whenever H is fixed, Robertson and Seymour have recently proven the following theorem [RS90, RS95]:

Theorem 1. *For any fixed graph H, there is an algorithm to decide if H is a minor of an input graph G that runs in time $O(n^3)$, where n is the number of vertices of G.*

This theorem states that the minor containment problem for any fixed graph H has polynomial-time complexity. Unfortunately, Robertson and Seymour's algorithm is not practical, since the actual polynomial-time bound has a large hidden constant.

Robertson and Seymour also showed that any family of graphs closed under the minor order (*lower ideal*) has a finite number of forbidden minors (*obstructions*) [RS85]. This follows immediately from the following theorem:

Theorem 2 (Graph Minor Theorem). *The minor order of graphs is a well partial order.*

For a lower ideal $\mathcal{F}$, if a finite set of obstructions $\mathcal{O} = \{O_1, O_2, \ldots, O_k\}$ is known, then the decision problem for $\mathcal{F}$ can be solved in a polynomial time using the set of obstructions. Here to check whether a graph G is in $\mathcal{F}$ one simply checks that each $O_i \in \mathcal{O}$ is not a minor of G; this procedure runs in time $O(kn^3) = O(n^3)$. However, this method is currently not practical because obstruction sets are frequently enormously large and, again, the only known polynomial-time algorithm (Theorem 1) has huge constants. Also finding the obstructions may be infeasible since the proof of Theorem 2 is non-constructive. An additional result by Bodlaender shows that there exists a linear-time algorithm (with fixed H) for the minor containment problem when the input has bounded treewidth [Bod93]. This algorithm also has extremely large constants. His theoretical result implies that if $\mathcal{F}$ excludes at least one planar graph then membership in $\mathcal{F}$ can be decided in linear time.

Fortunately, for many lower ideals with a large number of obstructions, it seems that a few of them are sufficient to detect most non-family graphs. For example, most non-planar graphs seem to contain $K_{3,3}$ as a minor. Thus almost all useful information comes from those "approximate" obstruction sets. (Langston and his students have done some work with the immersion order along these lines [Lan93, GLR94].) If we have fast (or feasible!) minor containment algorithms for a few select graphs, we have the potential to design simple heuristic algorithms for these lower ideals. We believe the simple general minor containment algorithm, presented in this paper, may be helpful in finding these approximating obstructions.

1.1 Preliminary definitions

We are now ready to formally define the graph problems of interest. In this paper only simple undirected graphs are considered (i.e., with no loops or multiple edges). A graph $G = (V, E)$ is represented as a finite set of vertices V and a set of edges E, where each edge is an unordered pair of vertices. Let u, v be any two vertices in a simple undirected graph G, we use uv to denote an edge between u and v in G.

For finite graphs there are many simple local operations that can be applied to change its structure. The following are some operations which are commonly used.

1. Delete an isolated vertex.

2. Delete an edge.

3. Contract an edge. In our definition, the contraction of an edge does not create loops or multiple edges.

4. Remove a subdivision. This is the same as contracting an edge incident to a vertex of degree 2 which is not contained on a cycle of length 3.

5. Lift an edge off a vertex. Here one replaces the two edges of a path uwv with a single edge uv.

The graph operations (3) and (5) are illustrated below in Figure 1.

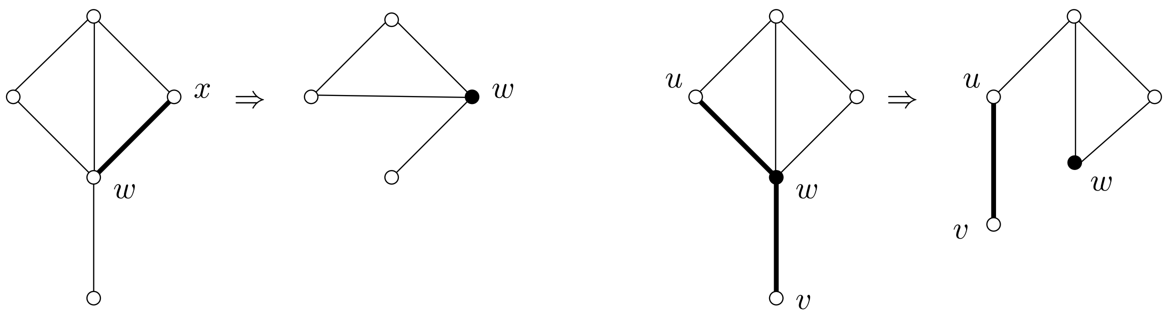


Figure 1: Contracting an edge xw of a graph G and lifting an edge off vertex w of G.

A *partial order* is a reflexive, transitive and antisymmetric binary relation. A *well partial order* is a partial order in which each infinite subset of its elements has at least two comparable elements. Four common graph partial orders are listed below.

Definition 3. *A graph H is a* subgraph *of a graph G if a graph isomorphic to H can be obtained from G by a sequence of operations chosen from (1) and (2). We let $H \leq_s G$ denote the* subgraph order.

Definition 4. *A graph H is a* minor *of a graph G if a graph isomorphic to H can be obtained from G by a sequence of a operations chosen from (1), (2) and (3). We write $H \leq_m G$ to denote the* minor order.

Definition 5. *A graph H is* topologically (homeomorphically) contained *in a graph G if a graph isomorphic to H can be obtained from G by the using the operations (1), (2) and (4). We write $H \leq_t G$ to denote the* topological order.

Definition 6. *A graph H is* immersed *in a graph G if a graph isomorphic to H can be obtained from G by the using the operations (1), (2) and (5). We write $H \leq_i G$ to denote the* immersion order.

Both the minor and immersion orders are well partial orders as proven recently by Robertson and Seymour [RS85, RSa, RSb]. The topological order is a well partial order for trees (see [Kru60]) but not, in general, for graphs. It is easy to see that the subgraph order is not a well partial order (e.g., consider the set of simple cycles). Thus, the minor (and immersion) order is a convenient partial order for giving structural characterizations of graph families. In other words, the Graph Minor Theorem implies that any set of graphs has a finite set of minimal elements in the minor ordering. Here a *finite set of minimal elements* is also called an *obstruction set*.

Below we give formal definitions about minor order lower ideals and obstructions.

Definition 7. *For two graphs G and H, a family $\mathcal{F}$ of graphs is a* lower ideal *(under the minor order) if $G \in \mathcal{F}$ implies that $H \in \mathcal{F}$ for any minor H of G.*

Definition 8. *A graph H is a* forbidden minor (obstruction) *for a lower ideal $\mathcal{F}$ if H is a minor-order minimal graph not in $\mathcal{F}$. That is, if we delete or contract any edge of H, then the resulting graph must be in $\mathcal{F}$. An* obstruction set *for a lower ideal $\mathcal{F}$ is the set of all the forbidden minors.*

We know that several popular graph families are minor order lower ideals, such as the planar graphs and those with small vertex covers and feedback vertex sets (see [CD94, CDF95]). Many of these lower ideals have been characterized by obstruction sets.

1.2 Graph containment problems

Our main objective is to design an useful minor order containment algorithm. We now formally list some partial order containment problems that correspond to the graph partial orders mentioned above.

Problem 9. **Minor Containment**
Input: Graphs $G = (V_1, E_1), H = (V_2, E_2)$.
Question: Does G contain a minor which is isomorphic to H, that is if a graph isomorphic to H is obtainable from G by the following operations: delete an isolated vertex, delete an edge, or contract an edge?

Problem 10. **Topological Containment**
Input: Graph $G = (V_1, E_1)$, $H = (V_2, E_2)$.
Question: Does G topologically contain a graph that is isomorphic to H, i.e. if a graph isomorphic to H is obtainable from G by the following operations: delete an isolated vertex, delete an edge, or remove a subdivision)?

Problem 11. **Subgraph (Isomorphism) Containment**
Input: Graphs $G = (V_1, E_1)$, $H = (V_2, E_2)$.

Question: Does G contain a subgraph isomorphic to H, i.e., a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \to V$ satisfying $uv \in E_2$ if and only if $f(u)f(v) \in E_1$?

Problem 12. **Immersion Containment**
Input: Graphs $G = (V_1, E_1)$, $H = (V_2, E_2)$.
Question: Does G contain an immersed graph isomorphic to H, i.e. if a graph isomorphic to H is obtainable from G by the following operations: delete an isolated vertex, delete an edge, or lift an edge)?

1.3 Outline of the paper

In this paper, we will focus on developing a simple usable algorithm for the minor order. Other partial order containment problems such as subgraph (isomorphism) containment and topological containment will also be discussed. We start our study with a simple subgraph containment algorithm. This is then followed in Section 3 with our development of a minor containment algorithm. Next in Section 4 we use this algorithm to generate a table of minor order counts. The last section concludes with some desirable areas for future work.

2 Subgraph Containment Problem

If a graph H is a subgraph of a graph G, then we know H is also a minor of G. Hence exploring the subgraph containment problem may be helpful in understanding the minor containment problem. There is a recently published algorithm for subgraph containment in [ESI98]. This algorithm is based on decomposing graphs to be matched into smaller subgraphs and has average computational complexity of $O(n^4)$ for a fixed H. More recently a linear-time algorithm for planar graphs G and H has been developed [Epp99]. Here, we only discuss a simple combinatorial algorithm to solve the subgraph containment problem which will be used as a foundation for our later developed minor containment algorithm. Before starting to discuss this algorithm, we define the notion of a vertex map.

Definition 13. *For two graph $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a* vertex map *is a function which maps V_1 to V_2. A vertex map $\mathcal{M}$ for graphs G_1 and G_2 is expressed as $\mathcal{M} : V_1 \to V_2$.*

We now mention a simple procedure to solve the subgraph containment problem for graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, where $|V_1| = n$ and $|V_2| = k$. To test whether H is a subgraph of G, we need to consider each subset V from V_1 such that $|V| = |V_2| = k$. With a one-to-one map chosen between V to V_2, if all edges in H exist in the induced graph $G[V]$ then we know that H is a subgraph of G.

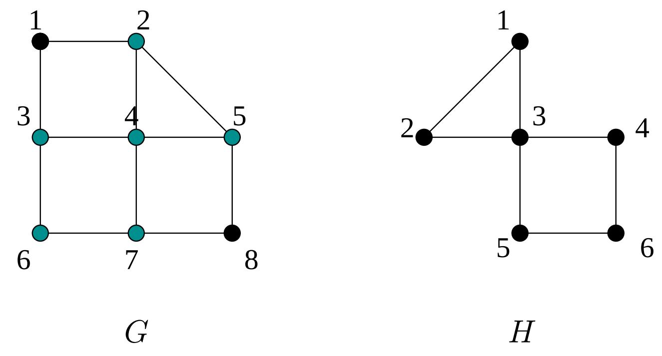


Figure 2: An example of the subgraph containment problem.

Figure 2 shows a vertex map between two graphs. Here $V_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $V_2 = \{1, 2, 3, 4, 5, 6\}$. If one chooses a subset $V = \{2, 3, 4, 5, 6, 7\}$ of V_1, a subgraph isomorphic to H is found by the one-to-one mapping between the vertices in V and the vertices in V_2 as follows:

$$\mathcal{M}(2) = 1, \mathcal{M}(3) = 4, \mathcal{M}(4) = 3, \mathcal{M}(5) = 2, \mathcal{M}(6) = 6, \text{ and } \mathcal{M}(7) = 5$$

As a summary, the pseudo code for an algorithm to solve the subgraph containment problem can be given as follows.

Procedure IsSubgraph(Graph $G = (V_1, E_1)$, Graph $H = (V_2, E_2)$)
    $n = |V_1|$
    $k = |V_2|$

    if $n < k$ **then return** `false`
    if $|E_1| < |E_2|$ **then return** `false`

    # Choose k vertices out of V_1 and store them in array V.
    # That is, we generate all $\binom{n}{k}$ combinations.
SetLoop: **while** next V from V_1 is obtained **do**

        # Generate all the permutations P of V in lexicographic order.
PermLoop: **while** next Permutation P of V is obtained **do**
            # Get a vertex map.
            for $i = 1$ **to** k **do**
                $M[V_2[i]] = P[i]$
            end for

EdgeLoop: **foreach** edge uv **in** E_2 **do**
                $w = M[u]$
                $x = M[v]$

```
                if not wx is an edge in E_1 then Next PermLoop
            end foreach

            return true
        end while
    end while
    return false
end
```

For this simple algorithm, the time bound in this algorithm is: $\binom{n}{k}k!|E_1||E_2|$. Thus when k is fixed, the time complexity for this algorithm is $O\left(\binom{n}{k}m\right)$, where $m = |E_1|$.

3 Minor Order Containment Problem

We now extend the previous method for solving the subgraph containment problem to partially solve the minor containment problem. We first restrict the host graph G to be a connected graph. Later in Section 3.6 we show how to handle arbitrary graphs.

For a fixed graph H, Robertson and Seymour gave an algorithm for the minor containment problem based on an algorithm for the disjoint connecting paths problem [RS85]. Although Robertson and Seymour proved that this algorithm runs in time $O(n^3)$, the actual running time is not very practical because of the large hidden constants. We plan to design a simple and practical algorithm (for arbitrary H) to solve the minor containment problem even though the time bound is greater than $O(n^3)$. To begin let us give an equivalent definition for the minor order.

Definition 14. *A graph H is a* minor *of a graph G if H is a subgraph of G', where G' is obtained by contracting edges from G.*

By this definition, a straight forward method is provided to decide if a graph H is a minor of a graph G, where G is a connected graph:

- **Step 1:** Generate all feasable vertex maps from G onto H.
- **Step 2:** For a vertex map, contract edges in G where the endpoints have the same image in H. This gives a resulting graph G'.
- **Step 3:** Test if $H \leq_s G'$. If it is, return true, otherwise iterate step 2.
- **Step 4:** Return false.

This method can only be applied for connected G since the order of G is reduced to the order of H only by edge contractions. The algorithm for disconnected input graphs can be solved easily by taking this algorithm as a subroutine.

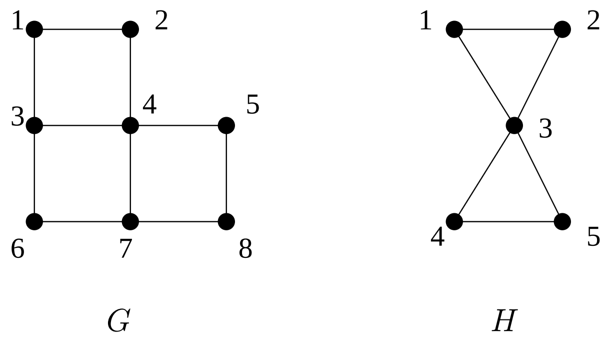


Figure 3: An example of a minor containment embedding.

The hardest part of this minor containment method is generating all vertex maps (as listed in Step 1), which we now explain how to do.

Let $G = (V_1, E_1)$ and $H = (V_2, E_2)$, where $|V_1| = n$ and $|V_2| = k$. For a connected graph G, the minor containment problem can be solved in time $O(k^n n^2)$ by checking all mappings from the vertices in G to the vertices in H, where subgraph containment for each mapping can be checked in $O(n^2)$ time. However, the time bound can be slightly reduced if only surjective vertex maps are generated. For example, if $V_1 = \{1, 2, 3, \ldots, n\}$ and $V_2 = \{1, 2, \ldots, k-1\}$. The two vertex maps $M_1 = \{1, 1, \ldots, 1\}$ and $M_2 = \{2, 2, 3, 3, \ldots, 3\}$ are impossible. Thus, in the next subsection, we discuss what is a necessary vertex map for solving the minor containment problem.

3.1 Vertex maps for the minor containment problem

We have discussed an algorithm to solve the subgraph containment problem. In this algorithm, each vertex in H is mapped to a unique vertex of G. For the minor containment problem, the number of vertices of G is greater than or equal to the number of vertices of H, that is $n \geq k$. Since extra vertices of G can be reduced by edge contractions, the vertex map for this problem is a many-to-one map (from G to H).

Figure 3 shows an example where $H \leq_m G$. A successfully mapping can be made with the mapping from G to H as follows.

$$\mathcal{M}(3) = \mathcal{M}(6) = 1, \mathcal{M}(1) = \mathcal{M}(2) = 2, \mathcal{M}(4) = 3, \mathcal{M}(7) = 4, \text{ and } \mathcal{M}(5) = \mathcal{M}(8) = 5$$

This example illustrates that for graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, a possible set of vertex maps has the following properties.

i. Each vertex in V_1 is map to only one vertex in V_2 (by definition of vertex map).
ii. Each vertex in V_2 must have a pre-image in V_1 under the mapping (i.e., the mapping must be surjective).

By these properties, in order to map vertices of V_1 to vertices of V_2, we can first partition V_1 to k non-empty disjoint subsets, then map each subset to a vertex in V_2. Therefore the combinatorial algorithm for generating all possible vertex maps is equivalent to the algorithm for generating set partitions of n with k blocks.

3.2 Set partitions with exactly k blocks

In this section, we discuss the k–block set partitions problem since the algorithm for generating all set partitions with k blocks is an important part of our minor containment algorithm. Our implemented algorithm to generate all set partitions with k blocks for a given vertex set will be based on these definitions (following the notation of [KS99]).

Definition 15. *A* set partition *of the set* $V = \{1, 2, \ldots, n\}$ *is a collection* $B_1, B_2, \ldots, B_j$ *of disjoint subsets of* V *whose union is* V*. Each* B_i *is called a* block*.*

For a given positive integer n, $\mathcal{S}(n)$ is used to denote the set of all partitions of $\{1, 2, \ldots, n\}$ into non-empty subsets. For positive integers n and k where $k \leq n$, $\mathcal{S}(n, k)$ is used to denote the set of all partitions of $\{1, 2, \ldots, n\}$ into exactly k non-empty subsets (set partitions of $\{1, 2, \ldots, n\}$ into k blocks).

Below we show all the partitions of the set $\{1, 2, 3, 4\}$, where periods separate individual sets.

1 blocks: 1234
2 blocks: 123.4 124.3 134.2 1.234 12.34 13.24 14.23
3 blocks: 1.2.34 1.24.3 1.4.23 14.2.3 13.2.4 12.3.4
4 blocks: 1.2.3.4

Each partition above has its blocks listed in an increasing order of each block's smallest element. A set partition can be encoded into a string called a "restricted growth string" based on this sort order.

Definition 16. *A* restricted growth string *(or RG string) is a string* $a[1..n]$ *where* $a[i]$ *is the block in which element* i *occurs.*

Define $\mathcal{R}(n)$ to consist of all restricted growth strings for set $\{1, 2, 3, \ldots, n\}$ and define $\mathcal{R}(n, k)$ to consist of all restricted growth strings corresponding to $\mathcal{S}(n, k)$.

Obviously and naturally, we have bijections between the sets $\mathcal{R}(n)$ and $\mathcal{S}(n)$ and between the sets $\mathcal{R}(n, k)$ and $\mathcal{S}(n, k)$. We can say a restricted growth string is another format of a set partition. Here are the RG strings corresponding to the partitions shown above.

1 blocks: 1111
2 blocks: 1112 1121 1211 1222 1122 1212 1221
3 blocks: 1233 1232 1223 1231 1213 1123
4 blocks: 1234

In the above table, $\mathcal{R}(4,3)$ is the third row.

Take two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$ where $n = |V_1|$ and $k = |V_2|$ with $k \le n$. To obtain a vertex map from G to H, we must first obtain a set partition with k blocks of V_1 and then map these k blocks onto the vertices of V_2. It is known that for positive integers n and k with $n \ge k$, the Stirling number (of second kind) is

$$S(n,k) = |\mathcal{S}(n,k)| = \frac{1}{k!}\sum_{j=1}^{k}(-1)^{k-j}\binom{k}{j}j^n \quad . \tag{1}$$

Therefore, to test all possible cases, our set partition algorithm must generate all those partitions $\mathcal{S}(n,k)$. In the next subsection, we will develop an algorithm for generating $\mathcal{S}(n,k)$ in restricted growth string format $\mathcal{R}(n,k)$ in lexicographic order.

3.3 An algorithm for generating $\mathcal{S}(n,k)$

We begin with an example. Let $n = 5$ and $k = 3$. Then by the above formula, $S(n,k) = 25$. All of the 25 RG strings for the set $\{1, 2, \ldots, n\}$ are given in Table 1 in lexicographic order.

Table 1: All set partitions of $\mathcal{S}(5,3)$.

No.	$\mathcal{R}(5,3)$	$\mathcal{S}(5,3)$	No.	$\mathcal{R}(5,3)$	$\mathcal{S}(5,3)$	No.	$\mathcal{R}(5,3)$	$\mathcal{S}(5,3)$
1	11123	123.4.5	10	12132	13.25.4	19	12313	14.2.35
2	11213	124.3.5	11	12133	13.2.45	20	12321	15.24.3
3	11223	12.34.5	12	12213	14.23.5	21	12322	1.245.3
4	11231	125.3.4	13	12223	1.234.5	22	12323	1.24.35
5	11232	12.35.4	14	12231	15.23.4	23	12331	15.2.34
6	11233	12.3.45	15	12232	1.235.4	24	12332	1.25.34
7	12113	134.2.5	16	12233	1.23.45	25	12333	1.2.345
8	12123	13.24.5	17	12311	145.2.3			
9	12131	135.2.4	18	12312	14.25.3			

The question we solve is: "How can our algorithm generate exactly those RG strings?"

The name "restricted growth" comes from the fact that RG strings are characterized by the following growth inequality:

$$a[i+1] \le (1 + max\{a[1], a[2], ..., a[i]\}) \text{ for } i = 1, 2, \ldots, n \text{ and } a[1] = 1 \quad . \tag{2}$$

When $\mathcal{R}(n,k)$ is taken into account, each $i = 1, 2, \ldots, k$ must occur at least once in a set partition. Thus we can use the above properties to design our algorithm.

We use $\mathcal{S}(5,3)$ as an example to illustrate the algorithm. We know the smallest RG string for $\mathcal{S}(5,3)$ is 11123 (from Table 1). This is the first RG string. The next string should be the smallest RG string which is greater than the previous one in lexicographic order. Let $a[1..n]$ be an integer array which stores the previous RG string. Once the previous string is generated, we can use the following method to find the next RG string.

- **Step 1:** Starting from the right most position of $a[1..n]$, use Formula 2 to find the lowest position pos in which $a[pos]$ can be increased by one. If $pos = 1$, then stop.

  For example, if $a[1..n] = 1,1,1,2,3$, then in this case pos is 3 because the position 3 is the lowest position in which $a[3]$ can be increased to 2 and the new string still satisfies Formula 2. This step ensures that the new RG string is greater than the previous one. In our example, after finishing this step, $a[1..n] = 1,1,2,2,3$.

- **Step 2:** Change each element which has more than one occurrence in the array to 1 in the remaining part of the array (i.e., from $a[pos+1]$ to $a[n]$).

  Thus in our example, the tail of the array $a[4..5] = \{2,3\}$. Let us use $occ[1..k]$ to count the number of occurrences for $i = 1,2,\ldots,k$ in the array $a[1..n]$. Since in our example, $occ[2] = 2$, then $a[4]$ can be changed to 1. But $occ[3] = 1$, so $a[5]$ is keep unchanged. Now we have the next RG string $a[1..5] = \{1,1,2,1,3\}$. This is the next string we want. But consider another example, if $n = 6$, $k = 4$, the previous string $a[1..6] = \{1,1,2,3,4,4\}$, then doing step 1 gives $a[1..6] = \{1,2,2,3,4,4\}$; doing step 2 gives $a[1..6] = \{1,2,1,3,1,4\}$. Whoops! Although this string is still a RG string, it is not the smallest one which is greater than the previous one. Thus to get the smallest RG string so far, we have to add one more step to sort the part of the array $a[(pos+1)..n]$ in increasing order to make it as small as possible.

- **Step 3:** Sort the part of the array from $pos+1$ to n in increasing order.

  This step ensures that the new RG string is the smallest one that is greater than the previous generated RG string.

Here, we give a formal object-oriented description of this algorithm. The following data structures and methods are used for implementing this algorithm.

- **Structure** SetPartitions
  To create a `SetPartitions` object, a positive integer n and a position integer k where $n \geq k$ must be taken as two parameters. The structure `SetPartitions` contains some data members:

SetPartitions(n, k)

An integer array which stores the next partition. Initially set to
$\{1, 1, \ldots, 1, 2, 3, \ldots, k\}$, the smallest lexicographic partition.
$partition[1..n]$

A boolean flag for stopping.
$finished$

End SetPartitions

- **procedure** Reset
  `Reset` is a procedure in the SetPartitions structure which initializes or resets the set partition to the smallest RG string.
- **procedure** NextPartition
  `NextPartition` is a procedure in the `SetPartitions` structure, which computes the next partition and stores it into the integer array *partition* and returns a boolean value to indicate if any more partitions are available. The algorithm is essentially implemented here.
- **procedure** Item
  `Item` is another procedure in the `SetPartitions` structure, which takes an integer i as input parameter and returns the ith element in the integer array *partition*. By combining procedure `NextPartition` and procedure `Item`, a new set partition in $\mathcal{S}(n, k)$ is obtained.

The algorithm for generating the set partitions with exactly k subsets (k blocks) of a given set $\{1, 2, \ldots, n\}$ is described as follows.

Procedure NextPartition()
  if $finished$ = `true` **then return** `false`

  # *occ* is an integer array, $occ[i]$ is used to store the number of occurrences of
  # $i = 1, 2, \ldots, k$ in the current partition.
  $occ[1..k] = \{0, 0, 0, \ldots, 0\}$

  # *max* is an integer array. $max[i]$ is the maximum number
  # in $partition[1..(i-1)]$ and $max[1] = 0$.
  $max[1..n] = \{0, 0, 0, \ldots, 0\}$
  $pos = n$

  # Compute $occ[i]$ where $i = 1, 2, \ldots, k$ for current partition.
  for i=1 **to** n **do**

```
        occ[partition[i]] = occ[partition[i]] + 1
    end for

    # Compute max[i] for i = 2, 3, ..., n
    for i=2 to n do
        if max[i − 1] < partition[i − 1] then
            max[i] = partition[i − 1]
        else
            max[i] = max[i − 1]
        end if
    end for

    # Find the lowest position pos in which partition[pos] can be increased by 1.
    while (pos > 1) and
        (partition[pos] = k or partition[pos] + 1 > 1 + max[pos]) do
        pos = pos − 1
    end while
    if pos = 1 then Stop!
        finished = true
        return false
    end if

    partition[pos] = partition[pos] + 1
    occ[partition[pos]] = occ[partition[pos]] + 1

    # Scan the subsequence of partition[pos + 1] to partition[n],
    # change every element to 1 such that this element has an
    # occ greater than 1.
    for i = pos + 1 to n do
        if occ[partition[i]] > 1 then
            occ[partition[i]] = occ[partition[i]] − 1
            partition[i] = 1
        end if
    end for

    # Sort the subsequence of partition[pos+1] to partition[n]
    # to make it smallest.
    Sort(partition[pos + 1], ..., partition[n])
    return true
end
```

Now, we prove that the above algorithm is correct.

Fact 17. *The procedure `NextPartition` within the `SetPartitions` structure is correct. That is, it successfully generates all set partitions with* exactly *k blocks for a* *given set* $\{1, 2, \ldots, n\}$*, where* n *and* k *are positive integers.*

Proof. To prove the algorithm is correct, we have to prove the following statements:

1. The algorithm will stop generating if no more partitions are available.
2. Each new generated partition is unique.
3. In the new partition, the number of occurrences of $i = 1, 2, \ldots, k$ must be at least 1.
4. The new partition, as a RG string, must be the next lexicographic RG partition string with k blocks.

We now prove that each new partition generated by the algorithm satisfies the above statements.

Statement 1: Since the new set partition is obtained by incrementing (by 1) the lowest position *pos* from the right. If *pos* is 1 (the leftmost position), this means the previous set partition can not be increased. Thus the algorithm sets a flag to indicate no more set partitions are available in this case. Thus statement (1) holds.

Statement 2: Since each generated partition is lexicographically greater than the previous one by increasing some position, according to the Formula 2, each partition is unique. Hence this statement holds.

Statement 3: In this algorithm, an integer array $occ[1..k]$ is used to count the number of occurrences for $i = 1, 2, \ldots, k$ in the new set partition. This ensures that the number of blocks (subsets) in a set partition is k. Thus, statement 3 holds.

Statement 4: Let *pos* be the position in which the previous set partition can be increased by 1. To ensure the new set partition is the smallest one which is greater than the previous partition, the remaining part of the new partition $p[(pos + 1)..n]$ must be assigned as small as possible in lexicographic order. The algorithm sets every element with more than one occurrence to be 1 (the smallest value) in this subsequence and sort the subsequence of $p[(pos + 1)..n]$ in increasing order. This ensures that the new set partition is the smallest set partition with k blocks in $\mathcal{S}(n, k)$ which is greater than the previous one. Thus any possible set partition in $\mathcal{S}(n, k)$ is not skipped. Hence, the statement holds.

Therefore, the algorithm for generating all set partitions with exactly k blocks for a given set $\{1, 2, ..., n\}$ is correct. □

In order to generate all set partitions, before calling the procedure `NextPartition`, we reset the array *partition* to be the smallest set partition $\{1, 1, \ldots, 1, 2, 3, \ldots, k\}$. By successively calling the procedure, all set partitions of $\mathcal{S}(n, k)$ will be generated in lexicographic order. The time to generate next set partition depends on the time of the sorting algorithm. If a linear-time bin-sort algorithm is applied, then this algorithm runs in time $O(n)$ per set partition.

We mention that this set partition algorithm was implemented and tested with the GNU EGCS-2.91 C++ compiler on an Intel Pentium 400MHz CPU. Table 2 shows some timing results based on this program.

Table 2: Timing results for generating set partitions ($n = 12$).

$\mathcal{S}(n,k)$	Time (secs)	$S(n,k)$ by algorithm and by Formula 1
$\mathcal{S}(12,1)$	0.00	1
$\mathcal{S}(12,2)$	0.00	2047
$\mathcal{S}(12,3)$	0.16	86536
$\mathcal{S}(12,4)$	1.12	611501
$\mathcal{S}(12,5)$	2.54	1379400
$\mathcal{S}(12,6)$	2.50	1323652
$\mathcal{S}(12,7)$	1.21	627396
$\mathcal{S}(12,8)$	0.32	159027
$\mathcal{S}(12,9)$	0.05	22275
$\mathcal{S}(12,10)$	0.00	1705
$\mathcal{S}(12,11)$	0.00	66
$\mathcal{S}(12,12)$	0.00	1

3.4 Implementing the minor containment algorithm (connected graphs)

In the previous section, we have developed a combinatorial set partition algorithm that generates all partitions with k blocks. Now we are ready to design an useful minor containment algorithm for connected input graphs. Recall from page 7 the four steps to decide, for two inputs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, where G is connected, if $H \leq_m G$. Let $n = |V_1|$ and $k = |V_2|$.

To implement step 1 (generation of all vertex maps from G onto H) we do the following. The set partition algorithm can be employed to partition V_1 into k blocks. And then one-to-one map each block into each vertex in V_2. Thus with respect to one set partition, there are $k!$ possible maps. Of course, all of them have to be tested. Thus the algorithm for generating all permutations of a set has to be applied. The C++ Standard Template Library (STL) contains a built in function that generates all permutations of a given integer set. Therefore, by applying our set partition generating algorithm and the standard permutation generating algorithm, all vertex maps between two graphs can be obtained.

Once step 1 is completed, the other steps are simple. For each map, we first pretest if those vertices in G which have the same image can be contracted (i.e., they form a connected subgraph). If it is false, then we skip to the next map. If it is true, then we check if each edge in H is embedded in the resulting graph G'. To do this, we do not have to do any edge contractions in G since this operation is very time consuming. Instead, we just test for each edge uv in H, if there is a corresponding edge wx in G, such that $\mathcal{M}(w) = u$ and $\mathcal{M}(x) = v$. If all edges in H can be tested successfully in this way, then the graph H is a minor of G and no more testing is required. Otherwise, after trying all vertex maps, we can conclude that H is not a minor of G.

This approach can be written as follows, where we assume `NextPermutation` is a build-in function which generates the permutations of a set.

```
Procedure IsMinor1(Graph G = (V1, E1), Graph H = (V2, E2))

    n = |V1|
    k = |V2|

    if n < k then return false
    if |E1| < |E2| then return false

    Make an initial SetPartitions object sps(n, k) for V1.

    # Permutations for V2 and vertex map.
    P[1..k]
    map[1..n]

setPLoop: while sps.NextPartition = true do
              # Initialize the permutation for V2 in P.
              for i = 1 to k do
                  P[i] = i
              end for

              # Pretest the set partition.
              for i = 1 to k do
                  Test if all of the vertices in G which
                      in the same ith block can be contracted
                      (i.e., these vertices are connected).
                  if it is not true then Next setPLoop
              end for

              # Get a map
permLoop:     while NextPermutation(P[]) = true do
```

```
            for i = 1 to n do
                map[i] = P[sps.Item[i]]
            end for

            for each edge uv in E_2 do
                if No edge wx can be found in E_1 where,
                    map[w] = u and map[x] = v then Next permLoop
            end foreach

            # We found the minor
            return true
        end while
    end while

    return false
end
```

Since our algorithm is derived from the definition of the minor order and we have proved that our set partition algorithm is correct, this algorithm is correct.

3.5 Complexity analysis

The running time of the algorithm depends on the number of partitions of n into k blocks, $S(n,k) = |\mathcal{S}(n,k)|$, and the number of permutations of V_2, where $|V_1| = n$ and $|V_2| = k$. Thus we have the following upper bound.

Lemma 18. *The running time of the minor containment algorithm* `IsMinor1` *is proportional to*

$$n^2|E_2|\sum_{j=1}^{k}(-1)^{k-j}\binom{k}{j}j^n \quad .$$

Proof. For two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, if $H \not\leq_m G$ then we need to search all possible $S(n,k)k!$ vertex maps. For each edge e in H, it takes time $O(n^2)$ to find if e is embedded in G. Thus it takes time $O(|E_2|n^2)$ to search all edges in H. Therefore, in the worst case, the running time for our minor containment algorithm to decide if $H \leq_m G$ is proportional to

$$n^2|E_2|S(n,k)k! = n^2|E_2|\frac{1}{k!}\sum_{j=1}^{k}(-1)^{k-j}\binom{k}{j}j^n k! = n^2|E_2|\sum_{j=1}^{k}(-1)^{k-j}\binom{k}{j}j^n,$$

where we used Formula 1 on page 10 for $S(n,k)$. □

When the graph H is fixed, k and $|E_2|$ are constants. Thus, we have the following.

Corollary 19. *The algorithm* **IsMinor1** *runs in time* $O\left(n^2 \sum_{j=1}^{k}(-1)^{k-j}\binom{k}{j} j^n\right)$, *whenever* H *is fixed.*

Although, the time bound is greater than $O(n^3)$ for a fixed H, it has a very small hidden constant. Therefore, for small simple graphs (which we are interested), this minor containment algorithm is useful. We compare the number of mappings required for a brute-force algorithm, k^n, against the set partition mappings required by `IsMinor1`, $S(n,k)k!$, for a few small ranges in Table 3. As n and k increase, there is a big difference.

3.6 Minor containment algorithm for disconnected graphs

For two graph G and H, our algorithm `IsMinor1` that decides whether $H \leq_m G$ only works if G is a connected graph. However, we can develop another algorithm which works whenever G is disconnected.

An algorithm `IsMinor2` for disconnected input graphs can be constructed as follows. Here we separate G and H into two sets of components C_G and C_H, respectively. If a graph H is a minor of a graph G then the set C_H needs to be mapped into C_G. The new algorithm `IsMinor2` simply calls `IsMinor1` for each possible mapping. Note that more than one component of H may map to the same component of G. Fortunately, the earlier algorithm `IsMinor1`, called as a subroutine, works in this case.

4 Minor Order Testing

In this section we use our minor containment algorithm `IsMinor1` to explore the popularity, as minors, of some small connected graphs. By "popularity" we mean how frequent a graph is a minor of a (possibly random) set of graphs. If the goal is to build an approximating membership algorithm for a (minor order) lower ideal, then these results give us some guidance on which subset of the family's obstructions to use.

We used our minor containment algorithm to obtain a table of popular graphs, which are independent of any particular lower ideal. The components of our testing program is as follows.

i. A minor containment algorithm (e.g., `IsMinor1`) for connected input graphs.
ii. A set of fixed graphs S_H.
iii. A set of connected input graphs for testing S_G.

This general program counts for each graph H in S_H how many of G in S_G are above H in the minor order. If the obstructions for a lower ideal are known and the

n \ k	2	3	4	5	6	7	8	9	10
2	2 4								
3	6 8	6 27							
4	14 16	36 81	24 256						
5	30 32	150 243	240 1024	120 3125					
6	62 64	540 729	1560 4096	1800 15625	720 46656				
7	126 128	1806 2187	8400 16384	16800 78125	15120 279936	5040 823543			
8	254 256	5796 6561	40824 65536	126000 390625	191520 1679616	141120 5764801	40320 16777216		
9	510 512	18150 19683	186480 262144	834120 1953125	1905120 10077696	2328480 40353607	1451520 134217728	362880 387420489	
10	1022 1024	55980 59049	818520 1048576	5103000 9765625	16435440 60466176	29635200 282475249	30240000 1073741824	16329600 3486784401	3628800 10000000000

$S(n,k)k!$
k^n

Table 3: Comparing the growth rate of $S(n,k)k!$ and k^n for $2 \le k \le n \le 10$.

set is small then we can easily replace the set S_H with it and possibly limit S_G to only non-family graphs.

For our experiment the set S_H is the set of all non-isomorphic connected graphs with orders 3 to 5. There are a total of 29 such graphs. The set of input graphs S_G was set of all non-isomorphic connected graphs with orders 3 to 9. Table 4 (starting on page 23) shows our computational result, where n is the order of the input graphs S_G and the row entries represent the number of times a graph (of S_H) is minor contained in the set of connected graphs.

From the computed table, it can be seen that those graphs with large size (number of edges) or large maximum degree are not popular.

5 Topological Order Containment Problem

In the previous two sections, algorithms are given to solve the subgraph containment problem and the minor containment problem. In this section, we briefly discuss a method for solving the topological containment problem. Although the topological order is not a well partial order in general, it is still very important in combinatorial graph theory. Since our main focus was on the minor order, here we only discuss one simple idea about solving the topological containment problem. Recall the definition of this problem: a graph H is *topologically contained* in a graph G if a graph isomorphic to H can be obtained from G by (1) deleting an isolated vertex, (2) deleting an edge, or (3) removing any subdivision.

The topological containment problem seems more complex than the minor containment problem because the edge contraction operation of the topological order is more restricted. Thus our minor containment algorithm can not be applied directly for the topological containment problem. To generate all graphs which are topologically contained in a given graph G, the following method may be used:

i. Taking a subgraph H of G.
ii. If removing some subdivisions of H gives T, then $T \leq_t G$.

Therefore a correct combinatorial algorithm to decide if a graph H is topologically less than a graph G is presented as follows.

Procedure IsTopologicalOrder (Graph G, Graph H)
  repeat
    Take a subgraph G' of G
    repeat
      Removing a set of subdivisions of G' yielding a graph T.
      if T is isomorphic to H **then return** `true`
      Restore G'

until no other set of subdivisions can be removed
until no other subgraph of G exists
return `false`
end

The algorithm `IsTopologicalOrder` clearly runs in exponential time since there are at least $\sum_{i=0}^{n-k} \binom{n}{i}$ possible subsets of vertices to consider when producing a subgraph G', where $|G| = n$ and $|H| = k$. We leave it as an open problem to find a more practical algorithm.

6 Conclusion

We have developed an usable algorithm for the minor containment problem. Although it is not theoretically the most efficient, it seems practical for graphs with about 10–15 vertices. The simple design, based on set partition maps, gives us faith in the correctness of the algorithm. Hence our generated tables provide a nice correctness check for future implemented algorithms.

Currently we are using the Graph Template Library (GTL) as our representation of graphs [FPR99]. Our experience says that we can probably increase the orders of graphs that `IsMinor1` can process by about five more vertices if we use a more specialized graph data structure.

We finish by mentioning some of the vast directions for future research. The next natural step would be to develop some "practical" polynomial-time minor containment algorithms where H is fixed. If this is too hard then one should try focusing on an algorithm for a class of graphs (e.g. planar graphs). In this respect, we should implement the known linear-time algorithms for bounded treewidth to see if they are practical (for small width). Here, an automated procedure could be invoked to produce an algorithm for each fixed graph H. Lastly, we should also address the possibility of designing feasible algorithms for the immersion and other partial graph orders, possibly using direct mappings as we did for our minor containment algorithm.

References

[Bod93] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the ACM Symposium on the Theory of Computing*, volume 25, 1993.

[CD94] Kevin Cattell and Michael J. Dinneen. A characterization of graphs with vertex cover up to five. In Vincent Bouchitte and Michel Morvan, editors, *Orders, Algorithms and Applications, ORDAL'94*, volume 831 of *Lecture Notes on Computer Science*, pages 86–99. Springer-Verlag, July 1994.

[CDF95] Kevin Cattell, Michael J. Dinneen, and Michael R. Fellows. Obstructions to within a few vertices or edges of acyclic. In *Proceedings of the Fourth Workshop on Algorithms and Data Structures, WADS'95*, volume 955 of *Lecture Notes on Computer Science*, pages 415–427. Springer-Verlag, August 1995.

[Epp99] David Eppstein. Subgraph isomorphism in planar graphs and related problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.

[ESI98] Yasser El-Sonbaty and M.A. Ismail. A new algorithm for subgraph optimal isomorphism. *Pattern Recognition*, 31(2):205–218, 1998.

[FPR99] M. Forster, A. Pick, and M. Raitner. The graph template library (GTL) manual. User manual, Universitat Passau, Germany, 1999. See `http://infosun.fmi.uni-passau.de/GTL`.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[GLR94] Rajeev Govindan, Michael A. Langston, and Siddharthan Ramachandramurthi. A practical approach to layout optimization. Technical report, Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996–1301, 1994.

[Kru60] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vaszsonyi's conjecture. *Transactions of American Mathematical Society*, 95:210–225, 1960.

[KS99] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, 1999.

[Lan93] Michael A. Langston. An obstruction-based approach to layout optimization. In *Contemporary Math*, volume 147, pages 623–629, 1993.

[RSa] Neil Robertson and Paul D. Seymour. Graph Minors. XX. Wagner's conjecture. in progress.

[RSb] Neil Robertson and Paul D. Seymour. Graph Minors. XXIII. Nash-Williams' immersions conjecture. in progress.

[RS85] Neil Robertson and Paul D. Seymour. Graph Minors – A survey. In *Surveys in Combinatorics*, volume 103, pages 153–171. Cambridge University Press, 1985.

[RS90] Neil Robertson and Paul D. Seymour. *An Outline of a Disjoint Paths Algorithm*, pages 267–292. 1990. *Algorithms and Combinatorics*, Volume 9.

[RS95] Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

Table 4: Number of minor containments in connected graphs of order $n = 3, 4, \ldots, 9$.

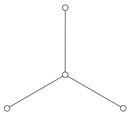
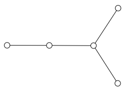
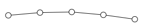
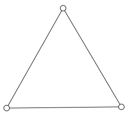
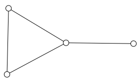
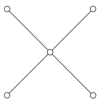
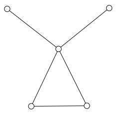
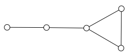
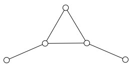
Graph n:	3	4	5	6	7	8	9	Total
Total number of connected graphs:	2	6	21	112	853	11117	261080	273191
1	2	6	21	112	853	11117	261080	273191
2	0	5	20	111	852	11116	261079	273183
3 	0	4	19	110	851	11115	261078	273177
4 	0	0	18	109	850	11114	261077	273168
5 	0	0	18	108	849	11112	261075	273162
6 	1	4	18	106	842	11094	261033	273098
7 	0	3	17	105	841	11093	261032	273091
8 	0	0	11	96	833	11091	261049	273080
9 	0	0	10	93	826	11073	261008	273010
10 	0	0	11	92	819	11055	260972	272949
11 	0	0	12	95	816	11033	260883	272839

Table 4 (continued): Number of minor containments in connected graphs.

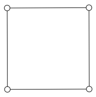
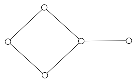
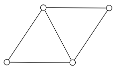
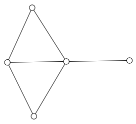
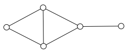
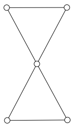
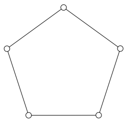
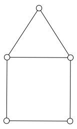
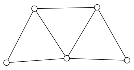
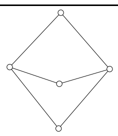
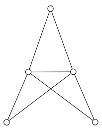
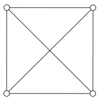
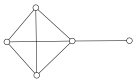
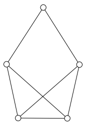
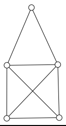
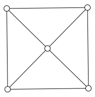
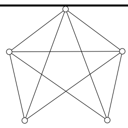
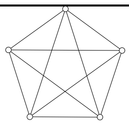
Graph n:	3	4	5	6	7	8	9	Total
Total number of connected graphs:	2	6	21	112	853	11117	261080	273191
12 	0	3	14	96	812	11011	260776	272712
13 	0	0	13	95	811	11010	260775	272704
14 	0	2	12	89	790	10929	260484	272306
15 	0	0	8	80	769	10866	260287	272010
16 	0	0	8	76	750	10799	260046	271679
17 	0	0	6	67	722	10718	259882	271395
18 	0	0	8	72	725	10669	259452	270926
19 	0	0	7	70	718	10642	259333	270770
20 	0	0	5	59	665	10382	258054	269165
21 	0	0	7	62	666	10277	257014	268026
22 	0	0	4	47	601	9965	255460	266077

Table 4 (continued): Number of minor containments in connected graphs.

Graph n:	3	4	5	6	7	8	9	Total
Total number of connected graphs:	2	6	21	112	853	11117	261080	273191
23 	0	1	6	56	612	9872	253898	264445
24 	0	0	4	49	600	9850	253858	264361
25 	0	0	5	52	597	9805	253575	264034
26 	0	0	3	40	545	9548	252257	262393
27 	0	0	3	32	442	8473	241874	250824
28 	0	0	2	23	362	7644	232819	240850
29 	0	0	1	9	164	4409	175313	179896