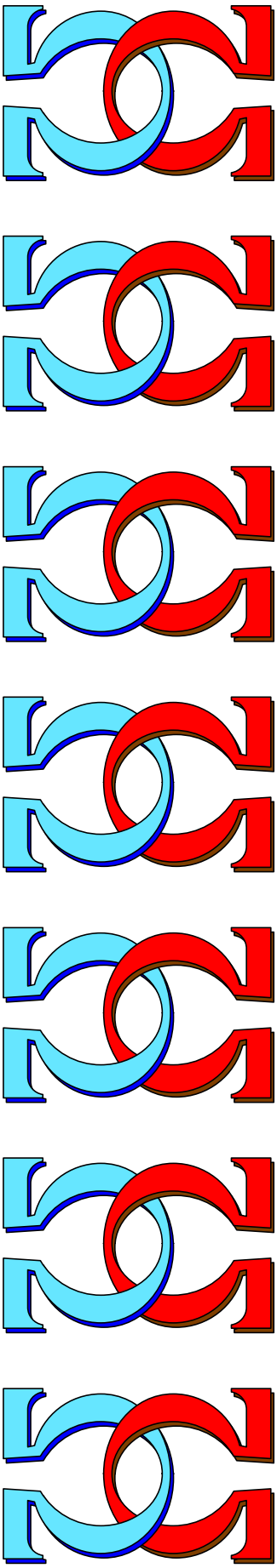


CDMTCS Research Report Series

Checking for Substructures in Graphs of Bounded Pathwidth and Treewidth

Jarrett Walsh
Armstrong Atlantic State University
Savannah Georgia, USA

Michael J. Dinneen
Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-205
December 2002

Centre for Discrete Mathematics and
Theoretical Computer Science

Checking for Substructures in Graphs of Bounded Pathwidth and Treewidth

Jarrett Walsh and Michael J. Dinneen
Department of Computer Science
University of Auckland, Private Bag 92019
Auckland, New Zealand

Abstract. Using a previously developed algebraic representation system for graphs of bounded pathwidth and treewidth, efficient membership algorithms are built for subgraph, minor and topological- minor partial orders. Semi-automatic procedures for these problems are presented through numerous explanatory figures and source code. An enumeration scheme for building membership automata for fixed graph substructures of graphs of bounded width will also be introduced.

1 Introduction

Graphs have found many uses throughout computer science, but many graph algorithms are impractical due to their complexity being $\mathcal{NP}$-complete [8, 14]. Examples of this type will be covered later; however, we want to focus on methods of dealing with such complexities and making restricted versions of these problems with algorithms that run in linear time. This solution lies in dealing with "bounded width" graphs. The goal of this paper is to provide practical solutions to the $\mathcal{NP}$-complete problems of determining subgraph, minor and topological minor family membership by restricting input to graphs of fixed width. From origins in basic graph theory, we will use the ideas of *pathwidth* and *treewidth* along with enumeration techniques to describe the problems [4, 5]. (The set of partial k-trees is equivalent to the set of graphs with treewidth at most k [1].) Starting with treewidth and pathwidth, we move into enumeration schemes and operations that are key to our algorithmic results given in Section 4. Here we will show how to generate, for input graphs of fixed width, in a semi-automatic way, linear-time algorithms (automata) for the subgraph problem and, in a specific case, the minor and topological order problems (see [12, 13, 16, 11]).

The focus of our studies is only on simple undirected graphs and the following conventions will be used. A *graph* $G = (V, E)$ consists of a finite number of *vertices* V and a finite set of *edges* E, where each edge is an unordered pair of vertices. The variable n (and sometimes $|G|$) is used to describe the *order* (the number of vertices) of the graph, while m is used to denote *size* (number of edges).

2 Treewidth and Pathwidth

The formal definitions and proofs in this section come from previous research given in [4, 5] and form the basis from which new studies stem. Treewidth and pathwidth play extremely important roles in keeping the complexity of the problems we deal with under control. A formal definition of treewidth (and pathwidth) follows, however an informal definition may be equally useful. A graph has width at most k if the vertices can be grouped into cliques of at most $k+1$ while creating a tree or path structure that follows certain rules.

1. Every vertex is in some clique.
2. If two vertices form an edge, they are members of a common clique.
3. For treewidth, the cliques of vertices (and edges) form a tree structure (i.e. no cycles).

The first definition of a tree decomposition is easily specialized for a path-structured case, therefore showing the link between treewidth and pathwidth. The formal definition for pathwidth follows theorems dealing with properties of treewidth.

Definition 1 *A* tree decomposition *of a graph $G = (V, E)$ is a tree T together with a collection of subsets T_x of V indexed by the vertices x of T that satisfies:*

1. $\bigcup_{x \in T} T_x = V$.
2. *For every edge (u, v) of G there is some x such that $u \in T_x$ include and $v \in T_x$.*
3. *If y is a vertex on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$.*

The width *of a tree decomposition is the maximum value of $|T_x| - 1$ over all vertices x of the tree T. A graph G has* treewidth *at most k if there is a tree decomposition of G of width at most k.* Path decompositions *and* pathwidth *are defined by restricting the tree T to be simply a path.*

Definition 2 *A graph G is a* k-tree *if G is a k-clique (complete graph) K_k or G is obtained recursively from a k-tree G' by attaching a new vertex w to an induced k-clique C of G', such that the open neighborhood $N(w) = V(C)$. A* partial k-tree *is any subgraph of a k-tree.*

Theorem 3 (see [3]) *The set of partial k-trees is equivalent to the set of graphs with treewidth at most k.*

The following paragraphs explain the idea of pathwidth and include several definitions and theorems that allow the building of graphs of fixed pathwidth and form the basis of an enumeration scheme vital to solving some complexity issues. To begin building graphs of fixed width, we must define a set of special vertices that the graph is built from.

Definition 4 *For a positive integer k, a* k-simplex *S of a graph $G = (V, E)$ is an injective map $\partial_S : \{1, 2, \ldots, k\} \to V$. A* k-boundaried graph *$B = (G, S)$ is a graph G together with a k-simplex S for G. Vertices in the image of ∂_S are called* boundary *vertices (often denoted by ∂). The graph G is called the* underlying graph *of B.*

Definition 5 *A graph G is a* $(k-1)$-path *if there exists a k-boundaried graph $B = (G, S)$ in the following family $\mathcal{F}$ of recursively generated k-boundaried graphs.*

1. *$(K_k, S) \in \mathcal{F}$ where S is any k-simplex of the complete graph K_k.*
2. *If $B = ((V, E), S) \in \mathcal{F}$ then $B' = ((V', E'), S') \in \mathcal{F}$ where $V' = V \cup \{v\}$ for $v \notin V$, and for some $j \in \{1, 2, \ldots k\}$:*
   (a) $E' = E \cup \{(\partial_S(i), v) \mid 1 \leq i \leq k$ and $i \neq j\}$, and
   (b) S' is defined by $\partial_{S'}(i) = \begin{cases} v & \text{if } i = j \\ \partial_S(i) & \text{otherwise} \end{cases}$.

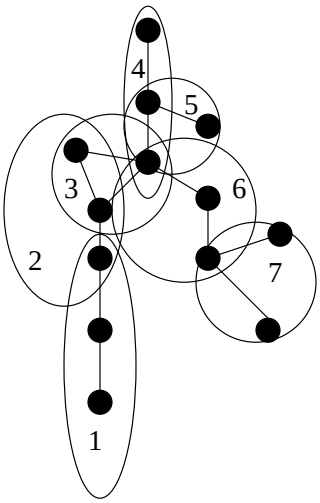
A partial k-path *is subgraph of a k-path.*

Now that a family of k-boundaried graph has been defined, it can be proved that a graph has pathwidth at most $k - 1$ if and only if it is a subgraph of an underlying graph of boundary size k. The formal definition of pathwidth now follows.

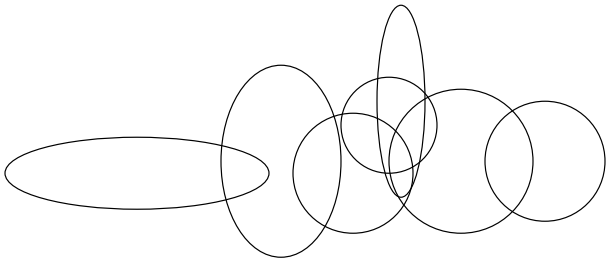
Definition 6 *A* path decomposition *of a graph $G = (V, E)$ is a sequence $X_1, X_2, \ldots, X_r$ of subsets of V that satisfy the following conditions:*

1. $\bigcup_{1 \leq i \leq r} X_i = V$.
2. *For every edge $(u, v) \in E$, there exists an X_i, $1 \leq i \leq r$, such that $u \in X_i$ and $v \in X_i$.*
3. *For $1 \leq i < j < k \leq r$, $X_i \cap X_k \subseteq X_j$.*

The pathwidth *of a* path decomposition *$X_1, X_2, \ldots, X_r$ is $\max_{1 \leq i \leq r} |X_i| - 1$. The* pathwidth *of a* graph *G, denoted $pw(G)$, is the minimum pathwidth over all path decompositions of G.*



A Graph of Pathwidth = 2, shown with cliques of size 3



Here, it is shown that the cliques induce a path where each point off the main path does not lead to another clique off the path (thus inducing a tree)

Figure 1: An example of a graph of fixed pathwidth.

The following definition (see [3]) defines a property of pathwidth that, when implemented, allows algorithmic results to be handled more easily.

Definition 7 *A path decomposition $X_1, X_2, \ldots, X_r$ of pathwidth k is* smooth*, if for all $1 \leq i \leq r$, $|X_i| = k+1$, and for all $1 \leq i < r$, $|X_i \cap X_{i+1}| = k$.*

From this, we see that a graph with a path decomposition of width k can be transformed into a smooth path decomposition of width k. The following lemma can also be generated from the definition.

Lemma 8 *If a graph G has components $C_1, C_2, \ldots, C_r$ then $pw(G) = \max_{1 \leq i \leq r} \{pw(C_i)\}$* .

One final lemma is necessary to qualify the use of pathwidth in defining family membership. It shows pathwidth is a *lower ideal* in the *minor order*, where $H \leq_m G$ if H can be obtained from G by a sequence of edge deletions, vertex deletions and edge contractions (see [1, 13, 16, 17]).

Lemma 9 (e.g. [12]) *If a graph H is a minor of G, $H \leq_m G$, then $pw(H) \leq k$.*

These are the basic theorems and definitions that explain the ideas of fixed pathwidth and treewidth. From here, an algebraic system of dealing with fixed boundary graphs will be necessary to explore other properties of graphs and graph families.

3 Graph Representation and Simple Graph Families

In dealing with graphs of fixed pathwidth, we use a finite number of boundary vertices. For all graphs of pathwidth k, the theorems in the last section tell us that we can generate any of those graphs from operators on a set of $k+1$ boundary vertices. In general, the boundary vertices are labeled from 0 to k in a k+1 boundaried graph. Defined below are several important operators that will allow us to construct the graphs within a fixed pathwidth family [4, 5]. (Note other representations are possible.)

3.1 Operators

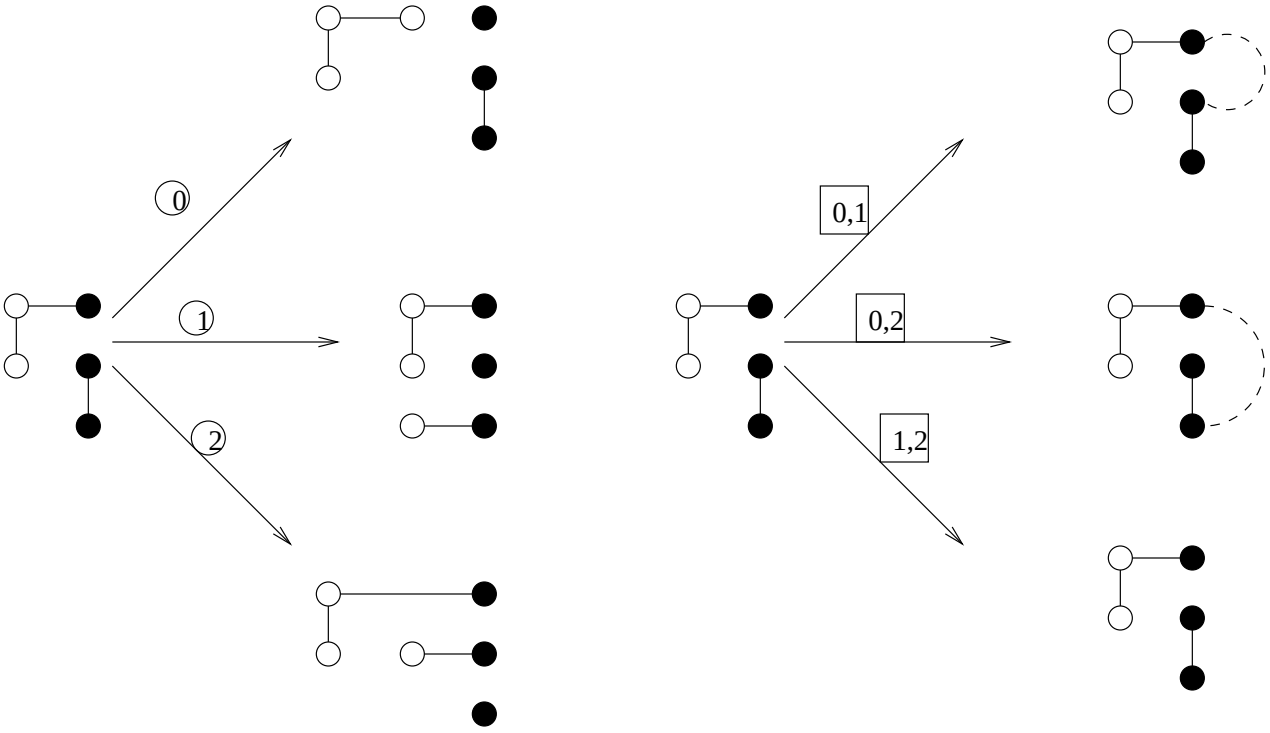
For a fixed boundary size, there are essentially three types of operators that can generate any graph in a "bounded-width" family. There are vertex operators, edge operators and, for the case of treewidth, a special operator called *circle plus*. The vertex operator is used to replace an existing boundary vertex with a new unconnected one. If the boundary vertex being replaced was connected to any other boundary vertices, it now becomes an interior vertex connected to the same boundary vertices as before. The edge operators are used to create a new edge between two boundary vertices. It is not necessary to create an edge between an internal vertex and a boundary vertex as this is possible with a combination of edge and vertex operators on the boundary.

Definition 10 *The edge operator,* $\textcircled{i}$*, places a new boundary vertex at position* i *in the boundary, forcing the old boundary vertex into the interior of the graph while keeping any connectivity to existing boundary vertices.*

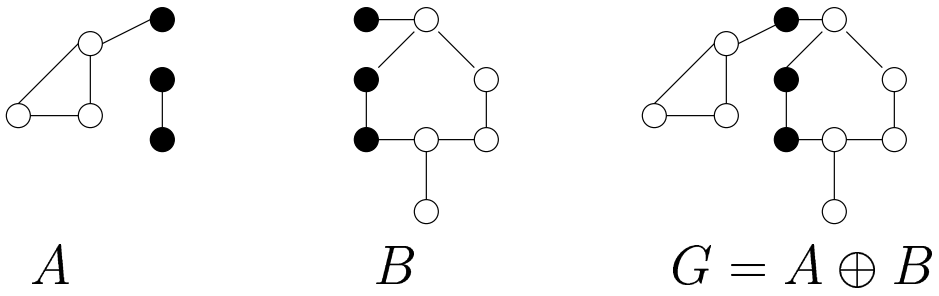
The vertex operator, $\boxed{i\ j}$*, places a new edge between two boundary vertices* i *and* j*. If the edge already exists, the edge operator does nothing.*

Two graphs with treewidth $k-1$*, of boundary size* k*, can be "glued together" with the* $\oplus$ *operator, called* circle plus*, that simply identifies vertices with the same boundary label.*

Example 1 *The edge operator and vertex operator is seen below on the same 3-boundaried graph. Vertex operators are used to create new, unconnected boundary vertices pushing the previous vertex into the interior of the graph. Edge operators add new connectivity on the boundary.*



The binary operator $\oplus$ on two 3-boundaried graphs A and B is illustrated below. Note that common boundary edges (in this case, the edges between boundary vertices 1 and 2) are replaced with a single edge in $G = A \oplus B$.



3.2 Paths and Cycles in Graphs

To begin exploring graph membership in families of fixed width, we start with small, simple cases before attempting complex membership algorithms for graph families of bounded width. These simple cases are necessary in that subgraphs, minors and topological minors are based on the simple cases like paths.

Definition 11 *Given a graph G, define* $\text{maxpath}(G)$ *as the length of the maximum path over all paths in G.*

Using the idea of maxpath, the idea of a *pathcover* can be established.

Definition 12 *Defined below is* pathcover(p). *A specific case,* 0*-pathcover is followed by the general definition.*

1. 0*-pathcover(p)* $= \{G \mid$ *maxpath(G)*$\leq p\}$
2. *k-pathcover(p)* $= \{G \mid$ *maxpath(*$G \setminus V'$*)*$\leq p$ *for some* $V' \subseteq V$*, where* $|V'| \leq k$ $\}$

A less formal explanation of a graph with k-pathcover(p) would be a graph that is within k vertices from having a maximum path of length p. By removing those k vertices, the longest of all paths left is still going to be less than p vertices. If we try to use k-pathcover(p) to define a family, we run across an item of interest.

Theorem 13 (see [4]) *k-pathcover(p) is lower ideal in the minor order, that is:*
For two graphs H *and* G *such that* $H \leq_m G$ *and* $G \in$ *k-pathcover(p),*
then $H \in$ *k-pathcover(p).*

Proof. Let G be a graph $\in$ k-pathcover(p), $G = (V, E)$ and $V' \subseteq V$. In taking the minor of a graph, there are only three operations: vertex deletion, edge deletion and edge contraction. The first two operators, by definition, can not increase a path since both operators inherently make paths shorter by decreasing order or size of a graph. The last operator may be less clear, thus a brief definition of edge contraction follows. Given two connected vertices: $e_{u,v} = (u, v)$ where vertices u and v may be connected to other vertices. Let u have edges $e_{u,a_1}, e_{u,a_2}, \ldots$ and $e_{u,v}$. Let vertex v have edges $e_{v,b_1}, e_{v,b_2}, \ldots$ and $e_{u,v}$. If an edge contraction on $e_{u,v}$ is done, vertices u and v are deleted and replaced by a new vertex, w such that: $e_{u,a_1}, e_{u,a_2}, \ldots = e_{w,a_1}, e_{w,a_2}, \ldots$ and $e_{v,b_1}, e_{v,b_2}, \ldots = e_{w,b_1}, e_{w,b_2}, \ldots$
For an edge contraction on $G \in$ k-pathcover(p), let $G' = G \setminus e_{u,v}$ represent the edge contraction of edge $e_{u,v}$. Because k-pathcover(p) removes a set of vertices V', the edge contraction may fall into this group of vertices being deleted. If both vertices of the contraction, u and v, fall into the set of vertices V' being deleted, the edge contraction will not occur. If one vertex, say u, is in V' then it does not matter the order of operations as the vertex u will be deleted on either operation. If u is deleted as part of V', no edge contraction will occur as the edge $e_{u,v}$ is deleted in the deletion of vertex u. Similarly, if the edge contraction removes u and v and replaces them with w that is connected to all previous connections of u and v, the connections of u would be lost in the removal of V' in such a way that $G' \setminus V' \subseteq G \setminus V'$ and maxpath(p)$\leq k$. If both vertices u and $v \nsubseteq V'$, then either operation can be done first with the same effect. Again, after edge contraction, G' will have lost at least one vertex in addition to those removed with V'. Since no new edges or paths were created in this deletion, maxpath($G' \setminus V'$)$\leq p$. $\square$

An interesting property of graphs in k-pathcover(p) families is that there is a bound on their pathwidth.

Theorem 14 (see [4]) *For any graph $G \in$ k-pathcover(p), pw(G) $\leq k + p$.*

Proof. We use a depth-first-search (DFS) tree to argue our case in this theorem. First look at a DFS spanning tree with root r of a graph G. Make a set of vertices for paths in the tree from root to leaf, that is for leaf v_i, the set $V_i = \{$vertices between r and $v_i\}$. By definition, the set $\bigcup_{n=1}^{i} V_i = V$. Since this is a DFS spanning tree, there are no cycles within the tree and therefore any edge on a path from r to v_i is fully in at least one V_i. All leaves are in specifically one V_i. To get pathwidth, we must show for any vertex, v and $i < k < j$, if $v \in V_i$ and $v \in V_j$, then $v \in V_k$ where $V_i \cap V_j \subseteq V_k$. If there is a vertex v that exists in both V_i and V_j, then it must be on the path from r to $v_{i,j}$ where $v_{i,j}$ is the closest vertex to v_i in the intersection of V_i and V_j. Since $v_{i,k}$ is visited after $v_{i,j}$ in the visit order of the tree, the path from the root to $v_{i,j}$ must be a subpath to the path from the root to $v_{i,k}$. Therefore $v \in V_k$ and $V_i \cap V_j \subseteq V_k$. Therefore, this is a valid method for tree decomposition. If the graph $G \in k$-pathcover(p), then it is within k vertices of having maxpath(p). Let W be a witness set of vertices such that maxpath($G \setminus W$)$\leq p$ where $|W| = k$. Deleting W leaves you with a new graph, G' whose maxpath is $\leq p$. Therefore, the maximum height of the DFS tree (the maximum path decomposition) is p. Therefore the pathwidth of G is at most $p + k$. □

Just as we defined maxpath(G) and k-pathcover(p), we can define the equivalent functions for cycles: maxcycle(G) and k-cyclecover(l).

Definition 15 *Let maxcycle(G) denote the length of the longest cycle in a graph G*

Definition 16 *k-cyclecover(l)= $\{G \mid \mid$ maxcycle$(G \setminus V') \leq l$ for some $V' \subseteq V$ where $|V'| \leq k\}$*

The idea of k-cyclecover(l) has specific graphical representations for values of l ranging from 0 to 2. These representations are described below.

k-cyclecover(0):	graphs with at most k vertices.
k-cyclecover(1) (i.e. k-vertex cover):	up to k vertices needed to remove all edges from a graph.
k-cyclecover(2) (i.e. k-feedback vertex set):	up to k vertices needed to remove all cycles from a graph.

Similar to the concept of k-pathcover(p), there exists a bound on the treewidth of those members of a family of k-cyclecover(l). Again, one can use DFS spanning trees to argue the point (see [4]).

Theorem 17 *The maximum treewidth of any $G \in k$-cyclecover(l)$= k + l - 1$.*

Proof. If a graph has maxcycle(G)$= k$, then in a DFS spanning tree, a cycle in the tree can only span $k-1$ levels. If the same procedure of placing paths from the root r to the leaves v_n is used, all vertices may be covered, but all cycle edges may not be included. For this reason, if each back cycle is placed into a unique V_i, a tree decomposition is formed that holds to all three tenants of treewidth. The first two properties are easily justified as all vertices and edges are included in some V_i. The final property of intersection of sets is forced to hold true due to the tree-nature of the DFS spanning tree. That is, for any two vertex paths V_i and V_j, there are no edges from V_i to V_j when $i \neq j$. Therefore, similar to our result with k-pathcover(p), we see that defining the sets gives a tree decomposition of treewidth at most $l - 1$. Since there is a witness W such that $W \subseteq V'$ where $|V'| \leq k$, and $G \in k$-cyclecover(l), maxcycle($G \setminus W$)$\leq k$, treewidth is at most $l + k - 1$. □

The concepts of maxpath and maxcycle were used in our study of graph membership, mainly as basic cases to build on. The further notions of pathcover and cyclecover, however, were not used. These last two functions, as we have seen are closely connected to pathwidth and treewidth and are lower ideals in the minor order as seen in the proofs. This theoretically allows the use of the algorithms introduced in the next section to be applied to obtain automata that will specify membership in families defined by these two operations.

One last item needs to be mentioned with respect to the complexity issues of checking for substructures of graphs that was earlier hinted at in the introduction. Two classic examples of $\mathcal{NP}$-complete problems is deciding membership in k-vertex cover and k-feedback vertex set for a graph G. The complexity arises when both G and k are part of the input. If the value of k is fixed, it can be shown that all "yes" instances have pathwidth or treewidth at most k. The complex problem has therefore been changed such that it is now tractable in the bounded width cases (see [6]).

4 Test Sets and Automata Generation

Introduced here is the idea of the *finite congruence test set*, a finite set of graphs that is representative of all graphs for a specific problem [7, 11, 8]

4.1 Test Sets

To begin to define test sets, we must first define a relation that puts the test set in perspective. Once the relation has been defined, an algorithm for generating the test set

will be introduced and proven.

Definition 18 *Define the congruence relation $\sim_f$ on the set of all graphs such that for graphs X and Y and a specific family of graphs, F:*
$X \sim Y$ if for all each boundaried graph Z, $X \oplus Z \in F \iff Y \oplus Z \in F$.

Furthermore, if X and Y are not congruent ($X \nsim_f Y$) a boundaried graph Z is called a distinguisher *if ($X \oplus Z \in F$ and $Y \oplus Z \notin F$) or ($X \oplus Z \notin F$ and $Y \oplus Z \in F$).*

A test set is the set of graphs, $\{Z\}$, that can distinguish membership into families and defines a congruence of classes of graphs. By generating a finite test set (i.e. a finite set T of tests such that if X and Y are not congruent then there exists a distinguisher $Z \in T$), we can start building graphs from the boundary and use the test set to see if operators on the graph lead to new states (equivalence classes). The only useful test sets are those that are finite.

The major work done for this paper was done on what we call a "kite family", based on the graph of a 3-cycle with a leaf, as seen in Figure 3. The test set algorithm about to be defined, however, is independent of the family of graphs being used, as long as they are of fixed pathwidth, and we have a function used to decide membership in the family. For explicit clarification, the functions used in this paper follow.

1. maxpath(G)
2. H a subgraph of G
3. H a minor of G
4. H a topological order of G

Definition 19 *Given a graph family F, and the equivalence relation $\sim_f$, a* test set *Υ_T is a set of boundaried graphs such that for fixed boundaried graphs X and Y:*
$X \sim_f Y \iff \{\forall T \in \Upsilon_T, Y \oplus T \in F \iff X \oplus T \in F\}$

The actual definition of the test set requires that the graph that your family is based on be subdivided into specific parts. An example of this is given in Figure 2.

Definition 20 *Define Υ_T by the following properties for a family based on a specific graph H (as in subgraph $\leq_s$, minor $\leq_m$ or topological minor $\leq_t$):*

1. *Label all vertices in H with i, b, or e denoting interior, boundary and exterior vertices.*
2. *Label all edges as i or e.*

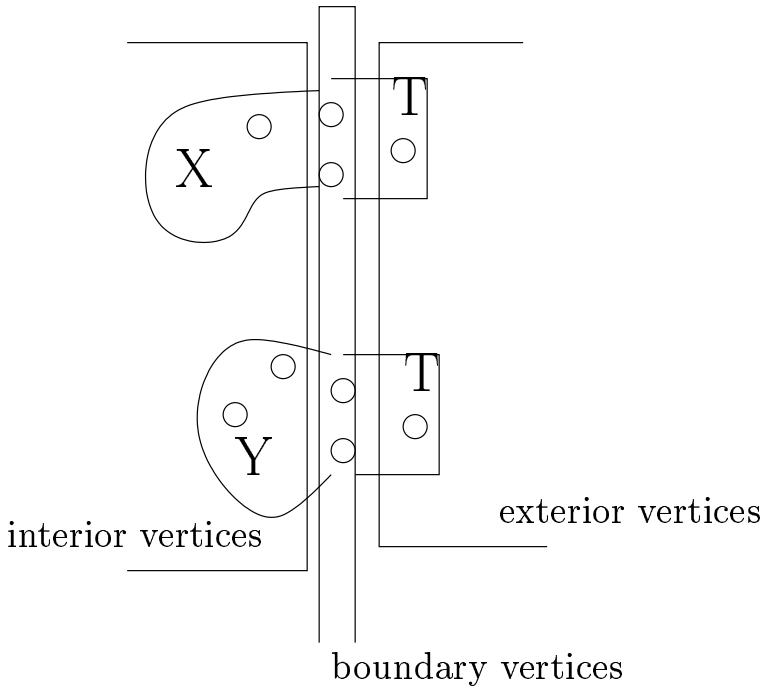


Figure 2: Boundaried graph split into interior, boundary and exterior parts.

Tests are derived from all permutations of the labeling when vertices and edges labeled i are removed from H leaving only b and e vertices and e edges.
If $(u, v) \in E(H)$, then vertexlabel(u)=vertexlabel(v) or vertexlabel(u)= b or vertexlabel(v)= b.
edge label $(u, v) = i$ if vertexlabel(u)= i or vertexlabel(v)= i
edge label $(u, v) = e$ if vertexlabel(u)= e or vertexlabel(v)= e
edge label $(u, v) = e$ or i if vertexlabel(u)=vertexlabel(v)= b
furthermore, there must be at least one vertex or edge labeled i.

Using this algorithm for creating test sets, we can generate a finite test set as we will now prove.

Proof. It suffices to show that if $X \not\sim_f Y$, there exists a $T \in \Upsilon_T(H)$ such that:
$X \oplus T \in H$ and $Y \oplus T \notin H$ or
$X \oplus T \notin H$ and $Y \oplus T \in H$.
Since $X \not\sim_f Y$, there exists a graph Z such that, without loss of generality, $X \oplus Z \in H$ and $Y \oplus Z \notin H$. Furthermore, let Z be minimal. That is, let Z be such that no $Z' \leq_s Z$ has the same properties of Z.
Consider all subgraphs S of $X \oplus Z = \{S_1, S_2, \ldots S_k\}$. If there exists an S_i containing

all of Z, then $Z \in \Upsilon_T(H)$. Otherwise, $Z \setminus S_i$ is nonempty for all i. Consider then $Z' = Z \setminus \{Z \setminus S_1\}$ which is a proper subgraph. Clearly, $X \oplus Z' \in H$ since it contains S_1. However, $Y \oplus Z' \notin H$ since $Y \oplus Z' \leq_s Y \oplus Z$ and $Y \oplus Z \notin H$. But $Z' \in \Upsilon_T(H)$ by the definition of $\Upsilon_T(H)$. □

With a well defined test set algorithm, the test sets generated are actually equivalence classes. This is easily seen in that if two states pass all the tests in a test set, they can be seen as equivalent. With a set of equivalence classes that define a family, the Myhill-Nerode theorem states that a deterministic finite automata is equivalent to the set of equivalence classes. Using this, we define a program that inputs the equivalence classes and creates the automata. The details of the Myhill-Nerode theorem can be found in the text by Hopcroft and Ullman [9]; the procedure to to generate automata for input graphs of bounded width was introduced in [7] and first implemented later in [4]. We can extend this notion to tree automata for graphs of bounded treewidth [15, 7].

4.2 Automata

Our first automata built was by hand, building a graph of maxpath(G)=2 with pw(G)=2. The resulting automata can be seen in Figure 4. After this case, the complexity of the automata increases dramatically, offering little hope to be seen in such a representation. With more states, there is also the issue of checking for congruent states, which, by hand, can take a great deal of time, when you are dealing with 57 test sets. There are methods, however, for having automata built through programming the test set[1]

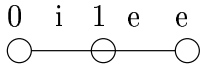
To build automata for membership in a family defined by an operation and a graph, the program required a fixed pathwidth as input, along with a membership function and the test set. The test sets, as mentioned are the essential part that set up the equivalence classes. The membership function is defined per case. For example, the two large automata that we ran were the families that contained the "kite graph" as a subgraph and then as a minor or topological minor. To describe the membership to the program, the kite was described in terms of a cycle of length three with at least one vertex with degree greater than 2. Using defined functions for looking up cycles and degree of vertices, the program was able to tell when the automata had built the desired graphs. See Example 2 for the membership function of the minor problem, a little bit more difficult to describe than the subgraph problem. In this case, not only do we need to find the subgraph, but if there is a cycle greater than 3 that has a chord or vertex in the cycle with degree greater than 3, it is now a member. The solution that was arrived

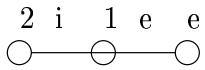
[1]We used adjacency matrices to store the boundaried graph tests, with the first t vertices denoting the boundary.

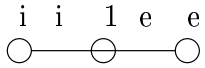
For maxpath(G)= 2 Problem

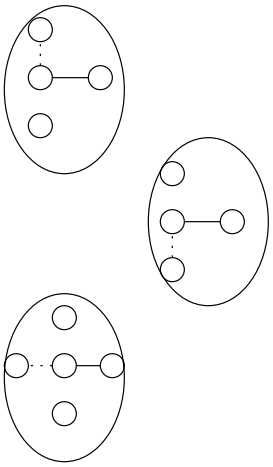
Vertex possibilities: {0,1,2}, i, e
Edge possibilities: i, e

Label all vertices in all possible permutations.



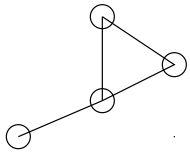




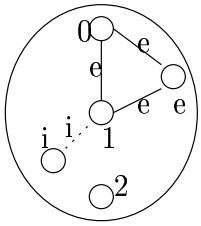


Relabling these permutations by deleting all interior "i" vertices and "i" edges gives the Tests.

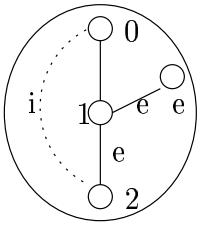
For the Kite Subgraph, Minor and Topological Minor Problem
Tests were clearly able to be divided into three groups, those with tails in the interior, the exterior or the boundary.



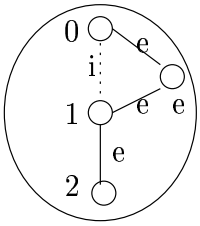
Kite Graph



tale internal



tale external



boundary tale

Following this scheme, we generate the following test set for kite subgraph where π n is the number of permutations of that type of graph over the boundary.

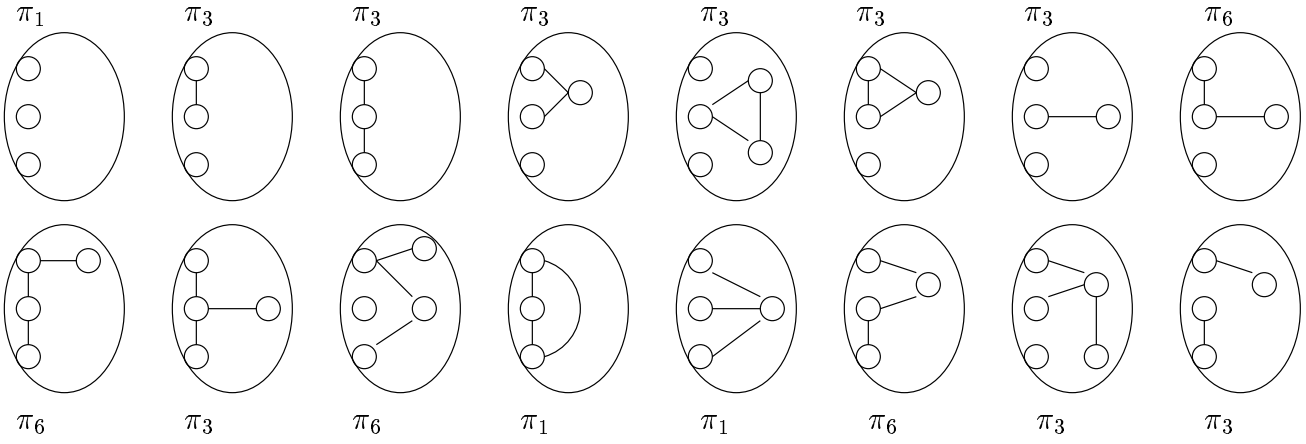
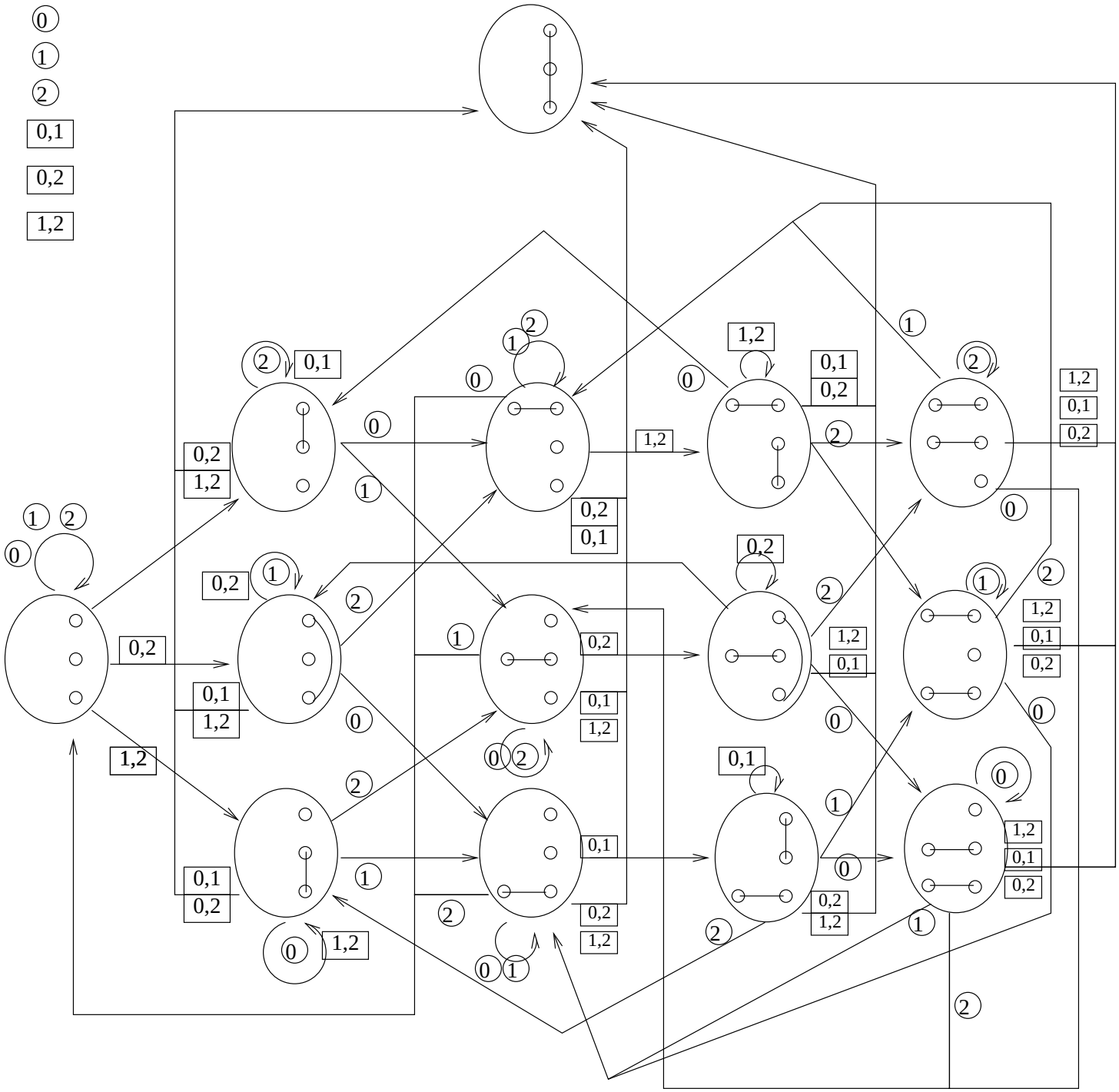
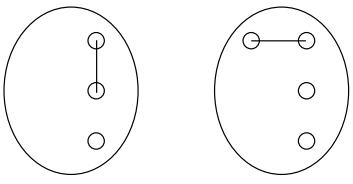


Figure 3: Test set example.



Generated with the following test set per the algorithm presented:



π_3 π_3

Figure 4: Automata for maxpath(G)=2 and pathwidth(G)=2.

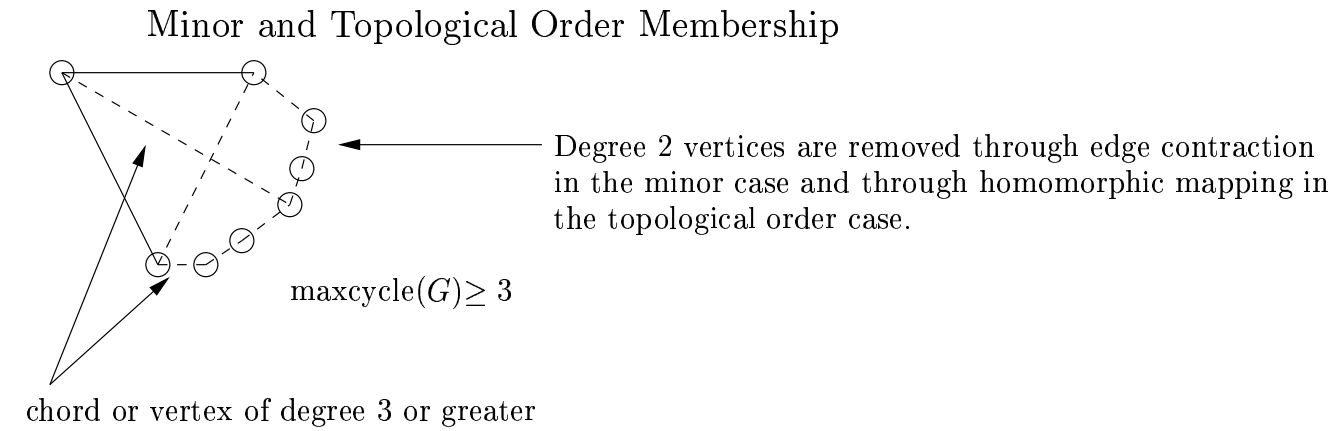
at was a depth first search tree that would allow the traversing of graph to look for those specific properties.

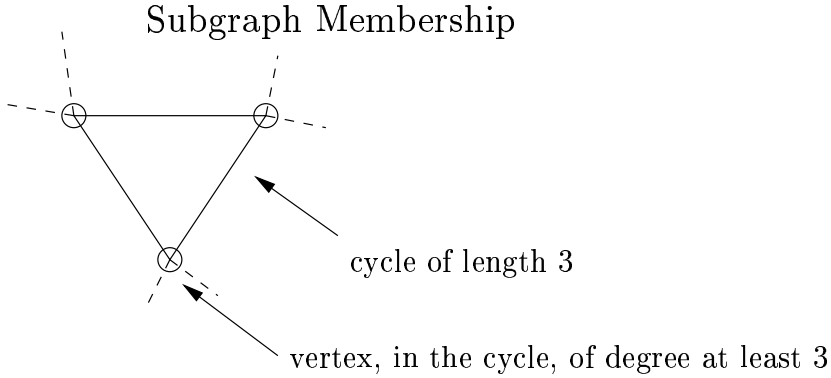
Example 2 `Bool has_kite(const Graph *G)`

```
{
    int n=G->vertices();
    if (n<4) return false;
    Bool answer = false;
    graph_int* deg = G->deg_vec(0);
    for (int i=0; i<n-2; i++)
    for (int j=i+1; j<n-1; j++)
    for (int k=j+1; k<n; k++)
    if (G->is_edge(i,j) && G->is_edge(i,k) && G->is_edge(j,k))
       if (deg[i] > 2 || deg[j] > 2 || deg[k] > 2)
       {
          answer=true; break;
       }
    delete deg;
    return answer;
}

void doDFS(const Graph *G, int root, int* parent)
{
    int n=G->vertices();
    for (int i=0; i<n; i++)
    {
      if (G->is_edge(root,i)==true && parent[i]==-1)
      {
         parent[i]=root;
         doDFS(G,i,parent);
      }
    }
    return;
}

Bool has_minor_kite(const Graph *G)
{
    int n=G->vertices();
    if (n<4) return false;
```

```
    Bool answer = false;
    graph_int* deg = G->deg_vec(0);
    int* parent =  new int[n];
    for (int i=0; i<n-2; i++)  // start dfs at vertex i;
    {
        for (int j=0; j<n; j++) parent[j]=-1;
        parent[i]=i;
        doDFS(G, i, parent);
        for (int u=0; u<n-1; u++) // check all edges in graph as DFS chords
        for (int v=u+1; v<n; v++)
        {
           if (G->is_edge(u,v) == false) continue;
           if (parent[u]>-1 && parent[v]>-1 &&
               parent[u] != v && parent[v] != u)
           {
             if (deg[u]>2 || deg[v]>2) { answer=true; goto stoplab;}
             int z = u;
             while (parent[z] != z)
             {
               z = parent[z];
               if (deg[z]>2) { answer=true; goto stoplab;}
             }
             z = v;
             while (parent[z] != z)
             {
               z = parent[z];
               if (deg[z]>2) { answer=true; goto stoplab;}
             }
           }
        }
    }
 stoplab:
    delete deg; delete parent;
    return answer;
}
```

With membership defined and equivalence classes processed, the output is a matrix form automata. The states are defined in terms of the edge and vertex operators, starting from the empty boundary and continuing until no new states exist. The states





and operator matrix for the kite subgraph, Figure 5 and 6 and the kite minor and topological order, Figure 5 and 5, can be found at the end, but one last interesting fact was noticed in preparing the test set for the 3 graphs. The test set for all three cases was identical. We look at the kite subgraph as the base case for this discussion. The kite minor would be defined as any graph with the kite as a subgraph or, after edge contraction and taking the subgraph, the kite was obtainable. Because the minor can undergo edge contraction, if any additional states were added to the subgraph test set, they would be equivalent after edge contraction to another member of the test set. The case was similar for the topological order in that any additional test sets could be reduced through the degree two contraction unique to that operation. One final interesting note is that the automata for the minor is identical to that of the topological order. The reason behind this is that their membership functions are identical. Figure 4.2 shows this concept clearly.

Through the proper graph enumerations leading to test sets, the Myhill-Nerode theorem can lead to computer-generated automata. The reason that these automata are so important is the fact that with an automata, the most complex problem becomes

solvable in linear time. There still exists research in this area in classifying the minor obstructions to the families that were addressed along the way and any others that can be generated through novel test sets.

5 Conclusion

In this paper we have shown, by example, how to build membership automata for various graphs families of bounded width. Our emphasis was on how to check if an input graph of bounded pathwidth (or treewidth) contains another fixed graph as a subgraph, minor or topological minor. We believe that our technique of using test sets for building membership automata for the illustrative simple "kite graph" can be easily extended for membership alogirithms for other small graphs. In the future, we hope to have a practical automated procedure for generating tree automata for checking for these types of embedded subtructures in graphs of bounded treewidth.

References

[1] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8:277–284, April 1987.

[2] 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.

[3] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. Preliminary manuscript, Dept. of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA Utrecht, the Netherlands, October 1995.

[4] Michael J. Dinneen. *Bounded Combinatorial Width and Forbidden Substructures.* Ph.D. dissertation, Dept. of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6, December 1995. (Library version January 1996).

[5] Michael J. Dinneen. Practical enumeration methods for graphs of bounded pathwidth and treewidth. Report CDMTCS-055, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, New Zealand, September 1997.

[6] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*, Springer-Verlag, 1999.

[7] Michael R. Fellows and Michael A. Langston. An analogue of the Myhill-Nerode Theorem and its use in computing finite-basis characterizations. In *IEEE Symposium on Foundations of Computer Science Proceedings*, volume 30, pages 520–525, 1989.

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

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory.* Addison-Wesley, Mass., 1979. Formerly titled *Formal Languages and their Relation to Automata* (1969).

[10] Jens Lagergren. *Algorithms and Minimal Forbidden Minors for Tree-decomposible Graphs.* Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, March 1991. Dept. of Numerical Analysis and Computing Sciences.

[11] Xiuyan Lu. Finite state properties of bounded pathwidth graphs. Master's project report, Dept. of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6, 1993.

[12] Neil Robertson and Paul D. Seymour. Graph width and well-quasi-ordering: a survey. In J.A. Bondy and U.S.R. Murty, editors, *Progress in Graph Theory*, pages 399–406. Academic Press, Inc., Toronto, 1984.

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

[14] Donald J. Rose. On simple characterizations of k-trees. *Discrete Mathematics*, 7:317–322, 1973.

[15] James W. Thatcher. Tree automata: an informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice-Hall, 1973.

[16] Liu Xiong and Michael J. Dinneen. The feasibility and use of a minor containment algorithm. Technical Report 171, Department of Computer Science, University of Auckland, Auckland, New Zealand, February 2000. Presented at the 5th Anniversary Workshop of CDMTCS (May 2000).

[17] Jan van Leeuwen. *Handbook of Theoretical Computer Science: A: Algorithms and Complexity Theory.* MIT Press / North Holland Publishing Company, Amsterdam, 1990. See "Graph Algorithms" chapter in pages 527–631.

[18] Herbert S. Wilf. Finite list of obstructions (the editor's corner). *American Mathematics Monthly*, pages 267–271, March 1987.

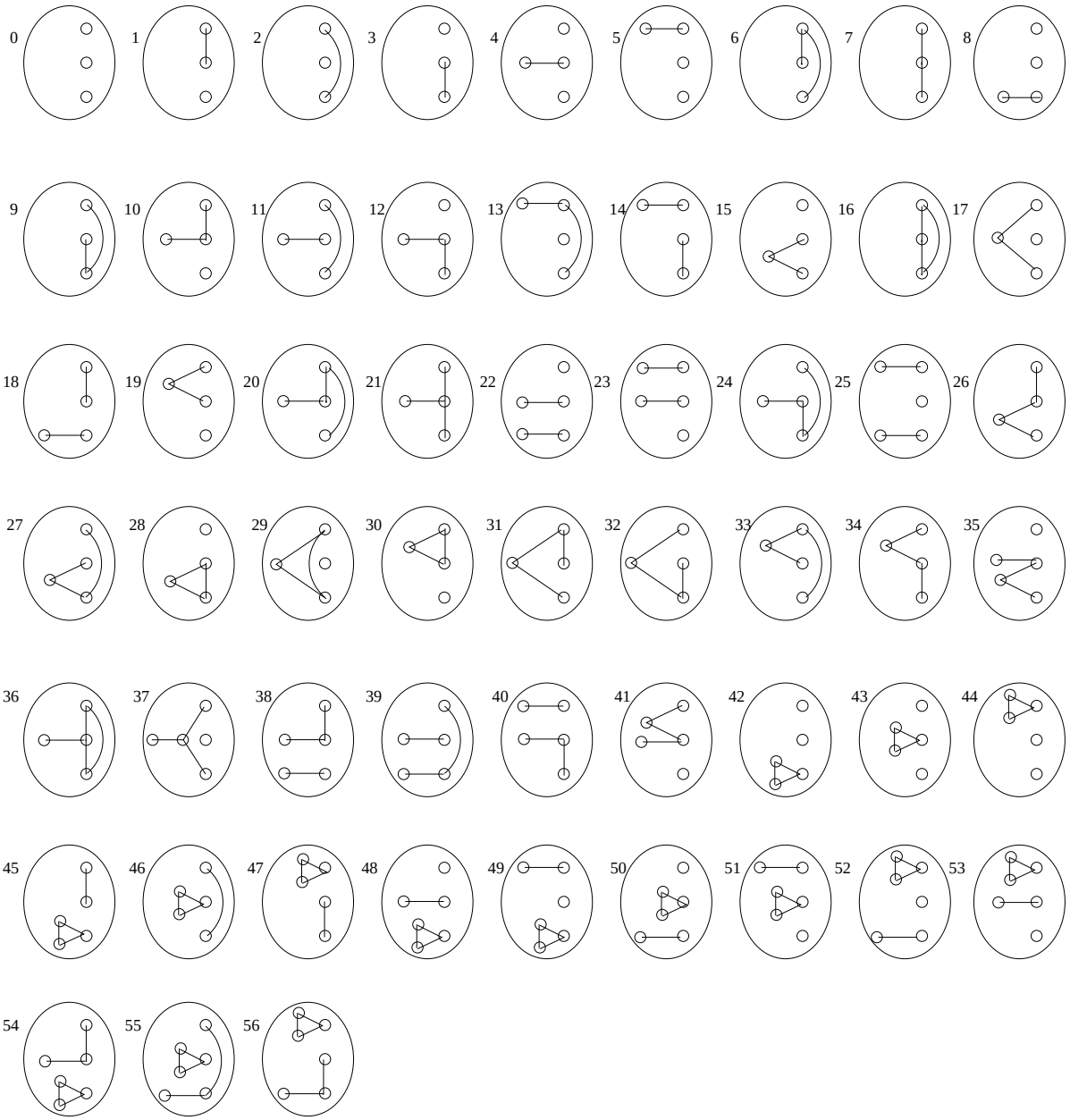


Figure 5: Subgraph membership equivalence classes for Kite graph.

0:	0	0	0	1	2	3
1:	4	5	1	1	6	7
2:	8	2	5	6	2	9
3:	3	8	4	7	9	3
4:	4	0	4	10	11	12
5:	0	5	5	10	13	14
6:	15	13	10	6	6	16
7:	12	17	10	7	16	7
8:	8	8	0	18	13	12
9:	12	13	19	16	9	9
10:	4	5	10	10	20	21
11:	22	2	23	20	11	24
12:	12	8	4	21	24	12
13:	8	13	5	20	13	24
14:	3	25	23	21	24	14
15:	15	8	4	26	27	28
16:	28	29	30	16	16	16
17:	8	17	5	31	29	32
18:	22	25	1	18	20	21
19:	4	5	19	30	33	34
20:	35	13	10	20	20	36
21:	12	37	10	21	36	21
22:	22	8	4	38	39	12
23:	4	5	23	10	39	40
24:	12	13	41	36	24	24
25:	8	25	5	38	13	40
26:	35	25	10	26	20	36
27:	35	13	23	20	27	36
28:	28	42	43	36	36	28
29:	42	29	44	36	29	36
30:	43	44	30	30	36	36
31:	22	37	10	31	36	21
32:	12	37	23	21	36	32
33:	22	13	41	36	33	24
34:	12	25	41	36	24	34
35:	35	8	4	26	27	36
*36:	36	36	36	36	36	36
37:	8	37	5	31	36	32
38:	22	25	10	38	20	21
39:	22	13	23	20	39	24
40:	12	25	23	21	24	40
41:	4	5	41	36	33	34
42:	42	42	0	45	36	36
43:	43	0	43	36	46	36
44:	0	44	44	36	36	47
45:	48	49	1	45	36	36
46:	50	2	51	36	46	36
47:	3	52	53	36	36	47
48:	48	42	4	54	36	36
49:	42	49	5	54	36	36
50:	50	8	43	36	55	36
51:	43	5	51	36	55	36
52:	8	52	44	36	36	56
53:	4	44	53	36	36	56
54:	48	49	10	54	36	36
55:	50	13	51	36	55	36
56:	12	52	53	36	36	56

Figure 6: The matrix columns, from left to right, are the three vertex operators, 0,1,2 followed by the three edge operators (0,1),(0,2) and (1,2). The values on the right are the states and correspond to Figure 5.

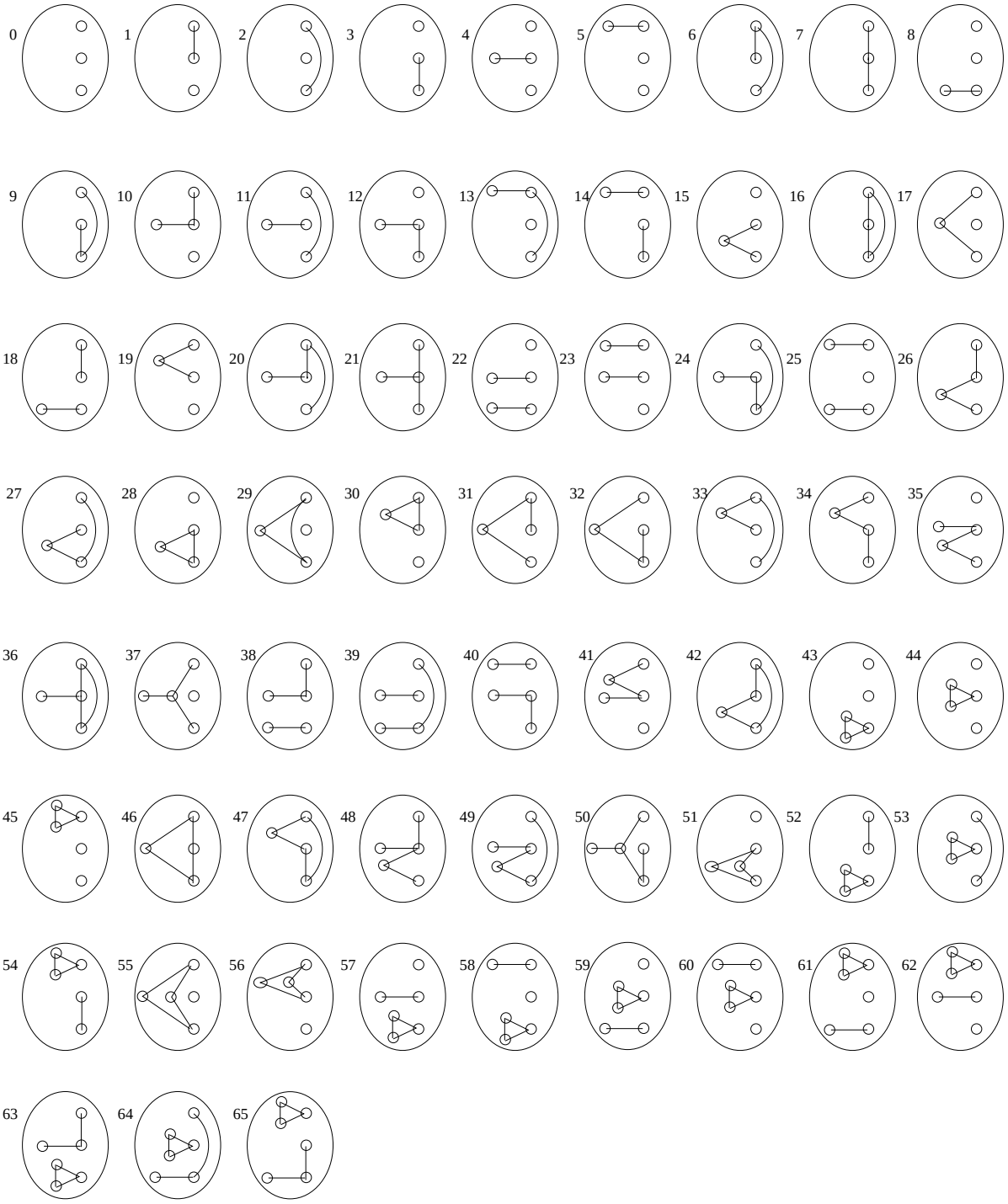


Figure 7: Minor-order membership equivalence classes for Kite graph.

0:	0	0	0	1	2	3
1:	4	5	1	1	6	7
2:	8	2	5	6	2	9
3:	3	8	4	7	9	3
4:	4	0	4	10	11	12
5:	0	5	5	10	13	14
6:	15	13	10	6	6	16
7:	12	17	10	7	16	7
8:	8	8	0	18	13	12
9:	12	13	19	16	9	9
10:	4	5	10	10	20	21
11:	22	2	23	20	11	24
12:	12	8	4	21	24	12
13:	8	13	5	20	13	24
14:	3	25	23	21	24	14
15:	15	8	4	26	27	28
16:	28	29	30	16	16	16
17:	8	17	5	31	29	32
18:	22	25	1	18	20	21
19:	4	5	19	30	33	34
20:	35	13	10	20	20	36
21:	12	37	10	21	36	21
22:	22	8	4	38	39	12
23:	4	5	23	10	39	40
24:	12	13	41	36	24	24
25:	8	25	5	38	13	40
26:	35	17	10	26	42	36
27:	35	13	19	42	27	36
28:	28	43	44	36	36	28
29:	43	29	45	36	29	36
30:	44	45	30	30	36	36
31:	15	37	10	31	36	46
32:	12	37	19	46	36	32
33:	15	13	41	36	33	47
34:	12	17	41	36	47	34
35:	35	8	4	48	49	36
*36:	36	36	36	36	36	36
37:	8	37	5	48	36	50
38:	22	25	10	38	20	21
39:	22	13	23	20	39	24
40:	12	25	23	21	24	40
41:	4	5	41	36	49	50
42:	51	29	30	42	42	36
43:	43	43	0	52	36	36
44:	44	0	44	36	53	36
45:	0	45	45	36	36	54
46:	28	55	30	46	36	46
47:	28	29	56	36	47	47
48:	35	37	10	48	36	36
49:	35	13	41	36	49	36
50:	12	37	41	36	36	50
51:	51	43	44	36	36	36
52:	57	58	1	52	36	36
53:	59	2	60	36	53	36
54:	3	61	62	36	36	54
55:	43	55	45	36	36	36
56:	44	45	56	36	36	36
57:	57	43	4	63	36	36
58:	43	58	5	63	36	36
59:	59	8	44	36	64	36
60:	44	5	60	36	64	36
61:	8	61	45	36	36	65
62:	4	45	62	36	36	65
63:	57	58	10	63	36	36
64:	59	13	60	36	64	36
65:	12	61	62	36	36	65

Figure 8: Minor-order membership automata for Kite graph.