Canonical Representations of Partial 2- and 3-trees

Stefan Arnborg
Andrzej Proskurowski

CIS-TR-89-11
July 11, 1989

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Canonical representations of partial 2- and 3-trees

Stefan Arnborg*
Andrzej Proskurowski†
Department of Computer and Information Science
University of Oregon, Eugene, Oregon 97403, USA

July 11, 1989

Abstract

We give linear time algorithms constructing canonical representations of partial 2-trees (series parallel graphs) and partial 3-trees.

1 Introduction

A canonical representation of a family of graphs assigns to each member of the family a label that is independent of any arbitrary vertex numbering: two graphs have the same canonical representation if and only if they are isomorphic. Thus, the graph isomorphism problem can be solved using canonical representations and solved efficiently if such representations can be efficiently computed and compared. Other uses of canonical representations are to investigate the structure of the automorphism group of a graph and to generate random graphs with some distribution over isomorphism classes.

Most graph representations are not canonical since vertices are arbitrarily numbered. But if we consider all possible vertex permutations, compute the corresponding representations, and select the lexicographically smallest, then we get a canonical representation. The set of vertex permutations yielding the lexicographically smallest

*Supported in part by a grant from NFR. Permanent address: NADA, KTH, S-100 44 Stockholm, Sweden.

†Research supported in part by the Office of Naval Research Contract N00014-86-K-0419.

representation is a coset of the automorphism group for the graph, regarded as a subgroup of the symmetric group on the vertex set.

A straightforward application of the above procedure has exponential (in the graph size) cost since there are exponentially many vertex permutations to minimize over. But in some cases it is possible to constrain the set of explicitly considered permutations in such a way that the whole procedure can be performed in polynomial time. We need only consider a set guaranteed to contain at least one coset of the automorphism group of the given graph. In this paper we show how these ideas yield an algorithm which produces a canonical representation for partial 3-trees in linear time, and thus also solves the isomorphism problem for partial 3-trees in linear time. Previously, the graph isomorphism problems for graphs of bounded valence (Luks [14]), genus (Filotti and Mayer [10], Miller [15], [16]), and tree-width (Bodlaender [6]) (none of which is a subfamily of another) have been shown solvable in polynomial time. Linear time algorithms for isomorphism of planar graphs (and thus also for partial 2-trees, which are planar) are already known (Fontet [11]; Hopcroft and Wong [13]; Colbourn and Booth [9]).

For a fixed value of the integer parameter k, partial k-trees are exactly subgraphs of chordal graphs with the maximum clique size $k+1$. Thus, partial 1-trees are the acyclic graphs (forests), and partial 2-trees are the series-parallel graphs (graphs with no K_4 minors or homeomorphs).

Partial k-trees have been in the focus of attention in recent years because of their interesting algorithmic properties. Namely, for a large number of inherently difficult (on general graphs) discrete optimization problems, partial k-trees admit a linear time solution algorithm when the value of k is fixed and the partial k-tree is given with its k-tree embedding. Somewhat discouraging is the fact that, for a general value of k, we do not know how to construct a k-tree embedding of such a graph in less than $\mathcal{O}(n^{k+2})$ time. The only more efficient and practical recognition (and embedding) algorithms known are for $k \leq 3$. A quadratic time recognition algorithm for any given k exists as a consequence of Robertson and Seymours results, but it uses a list of minimal forbidden minors which it is not known how to find and which can be of astronomical size.

The class of partial k-trees is also identical with the class of graphs of tree-width k (Scheffler [21], Wimer [25]). In the next section, we will give an iterative definition of partial k-trees that is the basis of our approach to solve problems for this class of graphs. Bodlaender proposed an algorithm for deciding isomorphism of partial k-trees [6]. His method is based on the brute force k-tree embedding method of Arnborg, Corneil and Proskurowski [3], where all k-vertex separators of the given partial k-tree are tested for suitability as separators in a k-tree embedding. This algorithm requires

solving bipartite matching problems and takes $\mathcal{O}(n^{k+4.5})$ time. We follow a different approach for $k = 2, 3$. Namely, for these values of k there exist small complete sets of safe reduction rules that determine k-tree embeddings of a given partial k-tree. With help of these reduction rules we produce a canonical string representing that graph in linear time, thus lowering the computation time from polynomial (actually, $\mathcal{O}(n^{7.5})$) to linear time for isomorphism of partial 3-trees.

The procedure we use is based on a canonical reduction sequence obtained from the safe reduction rules reported in Arnborg and Proskurowski [4]. For any given graph, the set of vertices reducible according to a given rule is fixed by the automorphism group of the graph[1]. Each reduction involves a separator of the graph with one, two or three vertices. Whether two reduced vertices are automorphic depends on symmetries between the corresponding separators. Our method keeps a record of symmetries of the reduced parts of the graph through a sequence of labels and orientations attached to the separators used in the reduction process. Two reduced subgraphs cut off by such separators are isomorphic (the isomorphism mapping one separator to the other) if and only if the labels of the two separators are equal, and their orientations indicate the correspondance between the separator vertex sets.

A reducible vertex represents a k-leaf in an embedding k-tree. Thus, adjacent vertices cannot be reduced in parallel. To deal with this, we refine the reduction rules to deal with overlapping reduction instances. These refined reduction rules allow us to construct a *parse tree*, where each node is associated with a reduction instance and two nodes are adjacent if the reduction instance corresponding to one creates a (hyper-) edge involved in the reduction corresponding to the other node. This tree is used to implement efficiently the algorithms computing canonical representations. In order to get linear time performance we use a technique of label numbering and bucket sorting described in Aho, Hopcroft and Ullman [1].

Our paper is organized as follows. After defining the necessary terminology in section 2 we introduce the method in section 3 by applying it to partial 2-trees. This special case is much simpler. Then the additional reduction instances necessary for partial 3-trees are derived in section 4. The algorithms used to give linear time performance are presented and analyzed in section 5.

2 Definitions and terminology

We will use standard graph theory terminology, as found, for instance, in Bondy and Murty [7]. We will also make use of concepts from the realm of hypergraphs, but will

[1]This means that the automorphism group of the graph permutes these vertices among themselves.

introduce them first in section 4. Some elementary and completely standard group theory is also used, see *e.g.*, Rotman [20]. We will now define some basic concepts.

A *walk* is a sequence of vertices such that every two consecutive vertices are adjacent. If all the vertices are different, we have a *path*. A walk forms a *cycle* if only its first and last vertices are identical. A set of k vertices, every two of which are adjacent, is called a k-clique. The graph on k vertices whose vertex set is a k-clique will be denoted K_k. A (minimal) subset of vertices of a graph such that their removal disconnects the graph is a (minimal) *separator*. A *k-tree* is a connected graph with no K_{k+2} subgraph such that every minimal separator is a k-clique. Equivalently, the complete graph on k vertices (K_k) is a k-tree, and any k-tree with $n > k$ vertices can be constructed from a k-tree with $n-1$ vertices by adding a new vertex adjacent to all vertices of a k-clique of that graph. In this new graph, the added vertex is a *k-leaf*. A *partial k-tree* is any subgraph of a k-tree.

While partial k-trees are undirected, simple graphs (without multiple edges or self-loops), in the course of our presentation we will allow both undirected *edges* and directed *arcs* (ordered pairs of vertices), as well as parallel edges and arcs. Those mixed graphs will be intermediate results of applying *graph rewriting rules*, consisting of replacing a subgraph isomorphic to the lefthand side of such a rule by the righthand side subgraph. In our case, the latter has always fewer vertices than the former and thus a set of such rules defines possible *reduction sequences*. Given a class of graphs, a rewriting rule such that its application preserves membership in both the class and its complement is called a *safe* rule. A set of reduction rules such that any non-trivial graph in the class contains as a subgraph the lefthand side of some rule is called a *complete* set of rules.

Complete sets of safe reduction rules for partial k-trees are known for $k \leq 3$ (Arnborg and Proskurowski [4]). Intuitively, they correspond to *pruning* of k-leaves in an embedding k-tree, safe in the sense of the existence of such a k-tree. For partial 1-trees (forests), the set of reduction rules consists of the removal of pendant vertices (of degree 1) and of isolated vertices. For partial 2-trees (series-parallel graphs) we have additional *series* and *parallel* rules, in which a path of degree 2 vertices is replaced by an edge, and multiple edges are replaced by one edge, respectively. For partial 3-trees, the additional rules deal with three cases of degree 3 vertex reductions: the *triangle*, the *buddy*, and the *cube* rules (*cf.* Figure 1). The cube-like configuration with the 'hub' (vertex x in Figure 1) of degree greater than 3 can be safely reduced, as well.

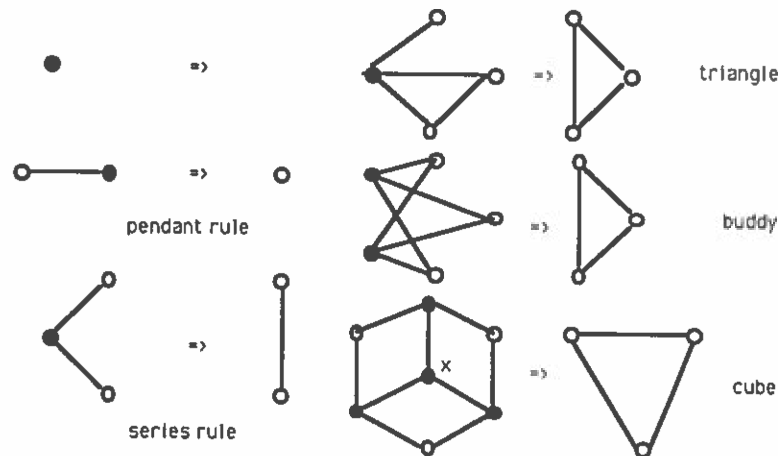

Figure 1: Reduction rules for forests, series-parallell graphs and partial 3-trees.

In a k-tree, vertices may be partially ordered with respect to the last $(k+1)$-complete subgraph ('the root') [17]. This partial order might be inherited by a partial k-tree, once we decide an embedding. Non-adjacent vertices reduced according to applicable reduction rules are not related in partial orders corresponding to embeddings having those vertices as k-leaves. Such reductions can be performed simultaneously (or, emulating simultaneity, consecutively in any order), leaving the necessary information as labels on other affected elements of the graph (for instance, on the pendant vertex in the pendant rule or on the added edge in the series rule).

Unfortunately, some instances of the reduction rules may deal with adjacent proposed k-leaves; obviously, for a k-tree embedding, only one of two such vertices is a k-leaf. This situation has to be dealt with separately. We note that there is a simple solution if the conflicting rules are different, *e.g.* a vertex reducible according to the triangle rule is adjacent to a vertex reducible according to the buddy rule. We simply order the rules and say that a higher priority rule takes precedence. The remaining case of conflict is where adjacent vertices are reducible according to the same rule. To break the ties in this case, we consider a refined list of rules derived from the complete set of safe rules presented in [4].

3 Canonical representation of partial 2-trees

We start our exposition by presenting the algorithm for partial 2-trees and then we generalize it to partial 3-trees.

Our algorithm is based on a construction of (vertex and edge) labels that record the sequence of reductions of the original graph G leading to a set of labeled isolated

vertices. From these, we construct the final label that is a canonical representation of G and, as such, would also allow a unique (and efficient) reconstruction of G.

Initially, every node label is (0) and every edge label is $(0,0)$. The second component in an edge label is the orientation information, locally recorded by 'an arrowhead' (say, an ordered pair of the edge's end vertices) or its absence. When recorded permanently, this information is transformed into a 0, a 1, or a -1, (like, for instance, in the label given above).

The algorithm will reduce a given connected graph to the single labeled vertex in phases performed in the following manner. Every phase begins with finding the first applicable reduction rule (closest to the beginning of the list of reductions, to be presented shortly). All instances of this rule in the entire graph are then found and the corresponding reductions are performed, resulting in a modified (reduced) graph, and some new labels. Since each reduction decreases the size of the graph, in $\mathcal{O}(n)$ phases the graph will be reduced to a single vertex.

3.1 Reduction rules

In this subsection we describe the rewriting rules (tailored to the needs of unique labeling) that reduce a given connected partial 2-tree to a single labeled vertex. The rules are given in their scanning order, together with the associated new label(s).

0 multiple vertices and edges. The rules below may create several edges with the same end-points (series rules) or several labels for a vertex (pendant and chain with identical end-vertices). The latter is referred to in the pendant rule 1 as a label being merged into the label of a vertex.

0.1 vertex label merge. The merging of multiple vertex and self-loop labels into a vertex label means that we keep a set of labels with multiplicities to describe the merged labels. Before the vertex itself is reduced, we must construct a proper label for it. This is done simply by lexicographically sorting the labels, and keeping duplicates in the sorted list l. The label will be $(0.1; l)$.

0.2 parallel rule. This rule is applicable when m edges have the same end-points s_1 and s_2 ($s_1 \neq s_2$, arbitrarily ordered). Let the labels of these edges be l_i and let d_i be 1 if edge i has an arrowhead from s_1 to s_2, -1 if an arrowhead is directed from s_2 to s_1, and 0 if it is not present. The parallel edges are replaced by one new edge between s_1 and s_2. This edge has a label $(0.2; l)$, where l is the lexicographically smallest of two sorted lists corresponding to edge orientations from s_1 to s_2 and from s_2 to s_1. The new edge will have arrowhead $(s_1 \rightarrow s_2)$ if

the first list is smaller, $(s_2 \rightarrow s_1)$ if the second is smaller, and none if they are equal. The first list has the members $(l_i, d_i)_{i=0}^{m-1}$, the second $(l_i, -d_i)_{i=0}^{m-1}$

1 pendant rule. This rule applies to vertices of degree 1 (pendant vertices) and to edges with both endpoints the same (self-loops).

1.1 dipole. If there is a pair of adjacent pendant vertices, they must form a connected component of the graph consisting of two vertices labeled l_1 and l_2 both incident to an edge labeled l', directed from the first to the second vertex $(d = 1)$ or not at all $(d = 0)$. This graph is reduced to a single vertex with label $(1.1; l_1, (l', d), l_2)$ or $(1.1; l_2, (l', -d), l_1)$, whichever is lexicographically smallest.

1.2 pendant vertices. Assume the set of pendant (degree 1) vertices is independent. One vertex and one edge is removed by a pendant vertex removal. The label l' together with with the orientation d of the edge form a pair (l', d). d is defined as 1 if the arrowhead of the edge is directed towards the pendant vertex, -1 if it is directed the other way, and 0 if it is not present. Combined with the label of the vertex, l, they form the label $(1.2; (l', d), l)$, which is merged subsequently into the label of the separating vertex according to rule 0.1.

1.3 self-loops. The label l' of a self-loop is merged into the label of its end-vertex according to rule 0.1.

2 series rule. This rule applies to vertices of degree 2. A set of such vertices inducing a connected graph will induce a path or a cycle in the graph, as detailed below:

2.1 chain rule. A maximal set of degree 2 vertices inducing a connected graph, and ending in two vertices, s_1 and s_2, of degree higher than 2 consists of a path $(v_1, \ldots, v_m)$ with v_1 adjacent to s_1 and v_m adjacent to s_2. Let l_i be the label of v_i, $i = 1, \ldots, m$ and l'_i of the edge (v_i, v_{i+1}), $i = 0, \ldots, m$, with $v_0 = s_1$ and $v_{m+1} = s_2$. Let d_i be 1, -1 or 0, depending on whether the arrowhead of (v_i, v_{i+1}) is directed from v_i to v_{i+1}, from v_{i+1} to v_i, or not present. The vertices v_i, $i = 1, \ldots, m$ and their incident edges are replaced by an edge connecting s_1 and s_2. The label of this edge is the lexicographically smallest of two labels, $(2.1; (l'_0, d_0), l_1, (l'_1, d_1), \ldots, l_m, (l'_m, d_m))$ and $(2.1; (l'_m, -d_m), l_m, (l'_{m-1}, -d_{m-1}), \ldots, l_1, (l'_0, -d_0))$. The arrowhead will point from s_1 to s_2 if the first alternative is smaller, from s_2 to s_1 if the second is smaller, and is absent if they are equal or if $s_1 = s_2$ (the latter condition describing a self-loop).

2.2 ring rule. A connected two-regular graph is a cycle consisting of vertices $v_0, \ldots, v_{m-1}$, with vertex v_i adjacent to vertex v_{i+1} for $i = 0, \ldots, m-2$ and with v_0 adjacent to v_{m-1}. It is reduced into a single vertex labeled with the lexicographically smallest of $2m$ labels $(2.2; (l'_{i+jk}, kd_{i+jk}), l_{i+jk})_{j=0}^{m-1}$ where $i = 0, \ldots, m-1$, $k = 1, -1$, and indices are computed modulo m.

3.2 Example

We will illustrate the process of creating a canonical label by reduction of the partial 2-tree whose adjacency lists are:

vertex	neighbors
1	13
2	13,14
3	14
4	15
5	6,15
6	5,7
7	6,15
8	12
9	10,12
10	9,11
11	10,12
12	8,9,11,13,15
13	1,2,12,14,15
14	2,3,13,15
15	4,5,7,12,13,14

The canonical reduction sequence produced is the following, where groups of parallell reductions are separated by horizontal lines:

rule	reduced	edges removed	label	labeled	arrow
1.2	1	(1,13)	l_1	13	
	3	(3,14)	l_1	14	
	4	(4,15)	l_1	15	
	8	(8,12)	l_1	12	
0.1			l_2	13	
			l_2	14	
			l_2	15	
			l_2	12	
2.1	2	(13,2), (2,14)	l_3	(13,14)	
	5,6,7	(15,5), (5,6), (6,7), (7,15)	l_4	(15,15)	
	9, 10, 11	(12,9), (9,10), (10,11), (11,12)	l_4	(12,12)	
0.1			l_5	12	
			l_5	15	
0.2		(13,14)	l_6	(13,14)	
2.1	12	(13,12), (12,15)	l_7	(13,15)	
	14	(13,14), (14,15)	l_8	(15,13)	$\rightarrow$
0.2		(13,15)	l_9	(13,15)	$\rightarrow$
1.1	13, 15	(13,15)	l_{10}		

The label abbreviations used above are as follows:

name	value
l_1	$(1.2; (0,0), (0))$
l_2	$(0.1; (0), l_1)$
l_3	$(2.1; (0,0), (0), (0,0))$
l_4	$(2.1; (0,0), (0), (0,0), (0), (0,0), (0), (0,0))$
l_5	$(0.1; l_2, l_4)$
l_6	$(0.2; (0,0), (l_3, 0))$
l_7	$(2.1; (0,0), l_5, (0,0))$
l_8	$(2.1; (0,0), l_2, (l_6, 0))$
l_9	$(0.2; (0,0), (l_8, -1), (l_7, 0))$
l_{10}	$(1.1; l_5, (l_9, -1), l_2)$

and we can get the explicit label by expanding the above abbreviations:

$$(1.1; (0.1; (0.1; (0), (1.2; (0,0), (1))), (2.1; (0,0), (0), (0,0), (0), (0,0), (0), (0,0))),$$
$$((0.2; (0,0), ((2.1; (0,0), (0.1; (0), (1.2; (0,0), (1))), ((0.2; (0,0), ((2.1; (0,0),$$
$$(0), (0,0)), 0)), 0)), -1), ((2.1; (0,0), (0.1; (0.1; (0), (1.2; (0,0), (1))),$$
$$(2.1; (0,0), (0), (0,0), (0), (0,0), (0), (0,0))), (0,0)), 0)), -1), (2.1; (0,0), (0), (0,0)))$$

4 Canonical representation of partial 3-trees

The idea behind the algorithm for partial 3-trees is similar to that of the algorithm constructing a canonical representation of partial 2-trees. The reduction of vertices is performed in consecutive stages, where each stage consists either of independent reduction instances or of groups of dependent reduction instances of the same kind. The situation is more complicated for partial 3-trees for two reasons. For one, the reduction information (recorded in labels) often concerns three vertices, and thus the resulting label must be associated with triples of vertices. We will present this as the labeling process for hyperedges of order 3. The second reason for a more complicated algorithm is the larger number of ways in which reduction rules of the same kind can involve adjacent vertices. For instance, more than one vertex of a triangle can have degree 3. Below, we elaborate on these differences leading to an algorithm constructing a canonical label for a given connected partial 3-tree.

4.1 Labeling of hyperedges

Each of the three reduction rules involving a degree three vertex causes the elimination of that vertex and edges incident with it and replaces them by edges between the neighbors of the vertex. (We call this reduction of a single vertex v adjacent to the separator vertices s_1, s_2, s_3 a *basic degree 3 reduction.*) We will separate this *topological* action from the *labeling* action of creating a labeled hyperedge corresponding to the three neighbors of the eliminated vertex.

Let a *sufficient set of permutations* denote permutations of vertices of a subgraph that are able to represent all symmetries (automorphism group) of the subgraph. (This can be achieved by a smaller than full symmetric group set of permutations when permutations exchanging non-symmetric vertices are ommitted.)

Let 0 be the label of hyperedges (vertices, edges, or triangles) in the original graph. A label of a hyperedge is determined with respect to a permutation of its vertices and is given an *orientation* represented by a subset of permutations of these vertices. These permutations are symmetric with respect to the label (they constitute the projection, into the symmetric group on the vertices of the hyperedge, of a coset of the automorphism group for the graph reduced into the hyperedge). When a hyperedge is removed in a reduction operation involving one of its vertices, its label will form a component of the label of the created hyperedge, together with an indication of how the orientation of the removed hyperedge corresponds to the orientation of the created hyperedge. Consider an orientation D of a removed hyperedge and a permutation σ of the union of vertices in the removed hyperedges. D is coded with respect to σ in the following way: Recall that D is a set of permutations of a subset of the vertices

appearing in σ, so each vertex occuring in D can be replaced by its index in σ. The resulting set of integer lists is then sorted lexicographically.

As an example, consider reduction of a degree 3 vertex v with neighbors a, b and c. This reduction removes at most one 1-hyperedge, three 2-hyperedges, and three 3-hyperedges. Now, for instance, the permutation $\sigma = (a, b, c, v)$ will cause the orientation $D = \{(a, b, v), (a, v, b)\}$ of a 3-hyperedge $\{a, b, v\}$ to be encoded as $((1, 2, 4), (1, 4, 2))$.

We can now describe the reduction of a set of vertices $R = \{v_i\}_{i=1}^{k}$ separated from the rest of the graph by vertices $S = \{s_i\}_{i=1}^{l}$:

Produce a sufficient set P of permutations of $R \cup S$. For each permutation p in P, produce a label as follows: consider the set of *(label,orientation)* pairs that contain some vertices of R. Replace each orientation with its encoding wrt p and sort the pairs to get a label wrt p. The subset of permutations that yield lexicographically smallest label l constitute the orientation of the new edge. The minimum value l and the rule number r according to which the reduction is made are used to build the label $(r; l)$ of the new hyperedge.

Observe that the previous label construction for partial 2-trees was only slightly different and can be easily changed to the present one. As an example, in the chain rule, each d_i would be changed from 1 to $((i, i+1))$, from -1 to $((i+1, i))$ and from 0 to $((i, i+1), (i+1, i))$. An arrowhead on an edge between vertices a and b would be represented as an orientation $((a, b))$ or $((b, a))$ and its absence would be represented by orientation $((a, b), (b, a))$.

4.2 Reduction rules for partial 3-trees

Except for applications of the parallel rules, the sequence of reductions made is decided solely on basis of adjacency information – we never investigate which type of hyperedge (2- or 3-hyperedge) makes two vertices adjacent. So in this section we can consider the graph represented by its clique representation when looking for vertex sets to reduce. After this set has been decided, we must use another data structure to find the set of hyperedges containing some reduced vertex, as detailed in section 5. The data structure for hyperedges is also used to decide when parallel rules (rule group 3) are to be invoked.

3 parallel triangles. This rule is applicable when two or more 3-hyperedges have the same set of vertices. Let the vertices be S=(s_1, s_2, s_3), and the labels of the hyperedges l_i, $i = 1, \ldots, m$. The orientations d_i, $i = 1, \ldots, m$ are sets of permutations of S. Let $d_i^{(p)}$ be the coding of d_i wrt a permutation p of S. For each permutation p of S, form the set $\{(l_i, d_i^{(p)}\}_{i=1}^{m}$, sort it lexicographically and

extract the lexicographically smallest list (over p) as well as those p giving a minimum, and call the smallest list l and the set of permutations P. The new hyperedge with vertices s_1, s_2, s_3 has label $(3; l)$ and orientation P.

4 isolated instances.

4.1 triangle. A vertex that is triangle-reducible and not adjacent to another triangle-reducible vertex can be reduced directly. This reduction of a single vertex adjacent to three separator vertices will be called the *basic degree 3 reduction.* The sufficient set of permutations consists of every permutation of the s_i, each followed by v.

4.2 buddy. Vertices in a buddy configuration not adjacent to another buddy configuration can be reduced simultaneously with the basic degree 3 reduction in all occurences.

4.3 cube. The three reducible vertices in a cube configuration not adjacent to another cube are reduced with the basic degree 3 reduction.

5 conflicting triangles.

5.1 diamonds. We can consider separately the cases when two adjacent triangle-reducible degree 3 vertices v_1, v_2 have two neighbors s_1, s_2 in common.

5.1.1 K_4. More than one occurrence of the diamond rule with adjacent reducible vertices is possible only if a connected component of the graph is the 4-clique K_4. The sufficient set consists of all 24 permutations of the vertices v_i, $i = 1, 2, 3, 4$.

5.1.2 K_4^-. If the vertices v_1 and v_2 are not adjacent, they induce, together with their common neighbors s_1, s_2, the four-clique without an edge, K_4^-. A sufficient set of permutations to consider is each of the two permutations of $\{s_1, s_2\}$ followed by each permutation of $\{v_1, v_2\}$.

5.2 Subgraph H. The remaining configurations of adjacent degree 3 vertices reducible according to the triangle rule and with at most one common neighbor (triangle-reducible for short) are discussed below. For a given partial 3-tree G, let us consider a maximal connected subgraph H consisting of triangle-reducible vertices subject to conflicting reductions according to the *triangle* rule. We will investigate the structure of those subgraphs and will make unique choices of *independently* reducible vertex sets in G. We consider the following cases of reductions in G depending on the degrees of vertices in H:

5.2.1 degree 1 only. Two adjacent vertices of degree 1 in H, v_1 and v_2, represent one of the two subgraphs in G shown in Figure 2. In the case of a four-vertex separator, other reduction rules must be applicable in the rest of the graph ('beyond the four separating vertices'), or else G cannot be a partial 3-tree. In the case of a 3-vertex separator, a separate reduction rule allows us to reduce v_1 and v_2, creating a hyperedge containing the separator vertices, with a unique label describing the reduced subgraph of the original graph (none of the separator vertices is triangle-reducible).

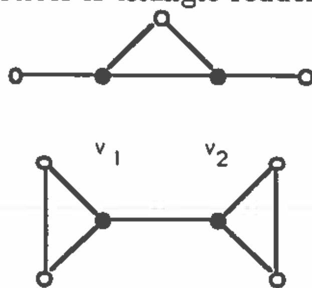

Figure 2: Subgraphs of H with two adjacent degree 1 vertices.

5.2.2 degree 1 and 2 or 3. Nonadjacent vertices of degree 1 in H can be subjects to the basic degree 3 reductions in G, since these do not conflict with each other.

5.2.3 degree 2 only. The vertices of degree 2 form a cycle in H. Let us call an edge t if it is a side of a triangle of G and f otherwise. Notice that end vertices of adjacent t edges have one common neighbor in G. Consider those vertices incident to both a t and an f edge; call this set A. Depending on the relation between A and H, we have three subcases (Figure 3).

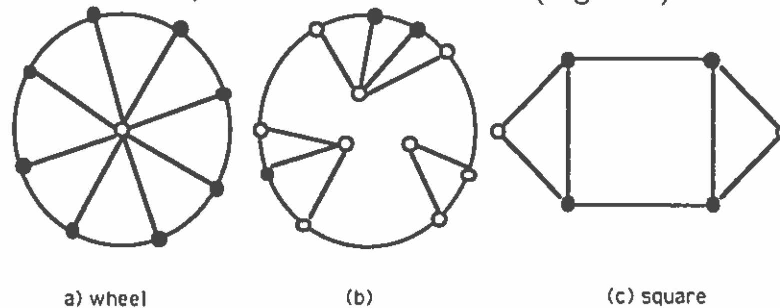

Figure 3: Cycles in H (a) a wheel, (b) a general case, (c) the square.

5.2.3.1 wheel. The set A is empty: all edges of H are t edges. This is the wheel configuration, reduced to a single vertex label ("the hub's") according to a separate rule.

5.2.3.2 collection of paths. A does not contain all vertices of H and its vertices partition the set of the remaining vertices of H into connected components. Those can be dealt with in a manner similar to that in rules 4.1, 5.2.1, and 5.2.2.

5.2.3.3 square. If H consists of alternating t and f edges (all the vertices are in A), then we have to consider subcases depending on the number of edges in H (it is trivially greater than 3). When H has 4 edges (only two triangles are involved), the triangles' 'third vertices' form a separator of G. The corresponding configuration (square, Figure 3(c)), is the left hand side of a separate reduction rule. If there are more than two triangles, then there must be another instance of a reduction rule 'beyond the separating vertices' (i.e., in the subgraph of G induced by vertices of G other than in H), or else G is not a partial 3-tree.

5.2.4 degrees 2 and 3. If H consists of both vertices of degree 2 and vertices of degree 3, then the vertices of degree 2 form in H paths that end in degree 3 vertices. When such a path has exactly two degree 2 vertices, these can be reduced according to rule 5.2.1. Otherwise, there are unique and nonadjacent vertices of degree 2 in H and the corresponding vertices in G can be reduced with the basic degree 3 reduction, similarly to the situation in 5.2.2.

5.2.5 degree 3 only (prism). Since a vertex in H is of degree 3 in the original partial 3-tree G, a cubic H is identical with G. To analyse this case, we consider the multigraph $\bar{H}$ obtained from H in the following manner: The set of vertices of $\bar{H}$ is the set of triangles (K_3 subgraphs) of H. The set of edges of $\bar{H}$ is the set of edges of H *not* in the triangles. A vertex of $\bar{H}$ is incident with an edge of $\bar{H}$ if the corresponding triangle contains a vertex of H incident with that edge in H. We will show that $\bar{H}$ is a series-parallel grah, which has important consequences in determining unique triangle reductions in H (or, equivalently, G). A multigraph is *series-parallel* if and only if it does not contain a subgraph homeomorphic to K_4. However, if $\bar{H}$ contains such a subgraph, then H contains as a minor the 4-regular Duffin graph (see Figure 4(b)), and is thus not a partial 3-tree. Since all vertices of $\bar{H}$ have degree 3, there must be instances of independent parallel edge reductions in $\bar{H}$. Unless H has six vertices and nine edges ('the prism', see Figure 4(a)) reduced to a single vertex by a separate rule, an instance of the parallel edge reduction in $\bar{H}$ corresponds to the subgraph of the *5.2.3.3 square* rule in H.

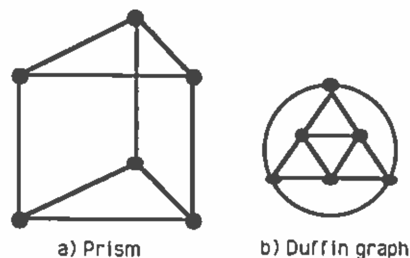

Figure 4: 3-regular subgraphs (a) the prism, (b) a minimal forbidden minor.

6 conflicting buddies. The 3-leaves v_1, v_2 in an instance of the buddy reduction can be adjacent to 3-leaves u_1, u_2 in another instance of the buddy reduction only if u_1, u_2 are commonly adjacent to a third vertex w. This third vertex may be identical with or different from the third common neighbor of v_1, v_2, leading to two configurations that can be incorporated as the left-hand-sides of new reduction rules. The former is a case of the *5.2.3.1 wheel* rule discussed above, the latter, *cat's cradle*, is shown in Figure 5.

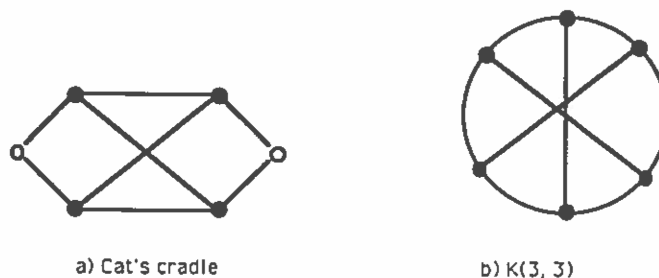

Figure 5: Overlapping buddy configurations.

6.1 $K_{3,3}$. Two cat's cradles can overlap only where the graph has a connected component which is $K_{3,3}$.

6.2 cat's cradle. This is the other case where $w \notin \Gamma(\{v_1, v_2\})$. The separator vertices s_1, s_2 and the cycle (v_1, v_2, v_3, v_4) have edges (s_1, v_1), (s_1, v_3), (s_2, v_2) and (s_2, v_4) between them.

7 conflicting cubes. There are two basic cases of possible conflicts between the reduced vertices in two different instances of the cube reduction.

7.1 qube. In one case, the purported 3-leaf vertices v_1, v_2, v_3 in one instance of the cube reduction are adjacent to 3-leaf vertices of another instance. This occurs only in the 8-vertex, 12-edge three-dimensional cube graph. This is because the

vertices u_1, u_2, and u_3 (*cf.* Figure 6) have then degree 3 and are adjacent to a common neighbor (the hub of that instance), also of degree 3.

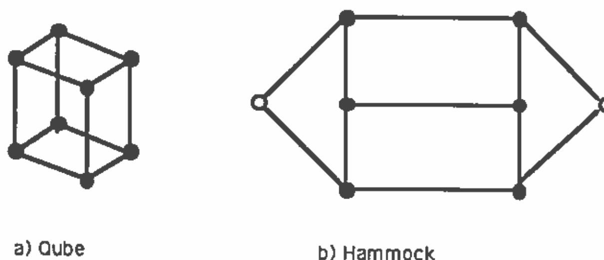

Figure 6: Overlapping cube configurations.

7.2 hammock. In the second case, the hub x of one instance is a 3-leaf of the other instance. This implies that one of the 3-leaves of the first instance, say v_3, is the hub of the second instance and its other neighbors u_1, u_2 are the remaining 3-leaves of the second instance. The vertices u_1, u_2 must have another common neighbor y. If this vertex is identical with the remaining vertex u_3 of the original cube, u_1, u_2 are triangle-reducible. If y is different than u_3, this leads to a new reduction rule with the left-hand-side configurations given in Figure 6(b).

5 Complexity of the algorithms

Any reasonable implementation of the labeling and reduction process will run in polynomial time for some low-order polynomial. In order to get a *linear time* algorithm we must be more careful. Some apparent obstacles to linear time performance are introduced by the need to perform the following tasks:

(i) Sorting $\mathcal{O}(n)$ labels, each of length $\mathcal{O}(n)$, when producing labels for applications of parallel rules.

(ii) Finding the lexicographically smallest of m labels each of size k, where mk is $\mathcal{O}(n)$, when deciding label for a 2.2 (ring) reduction.

(iii) Finding instances of applicable reductions.

Actually, there are already methods available to overcome these obstacles. First, the *abbreviations* introduced in the example of section 3.2 can be formalized into a *canonical numbering* of labels (or objects labeled), as is done for a tree isomorphism algorithm due to Edmonds and described in, e.g., Aho, Hopcroft and Ullman [1] and

Colbourn and Booth [9]. The technique presented there is directly applicable here, and solves problem (i). With this canonical numbering, any of the corresponding algorithms by Syslo [23], Shiloah [22], or Booth [8] solves also problem (ii) above.

The total number of times vertex adjacencies change during the reduction process is proportional to the graph size. Thus, maintaining 'ready lists' for vertices that reach small enough degree to be considered in the reduction rules resolves problem (iii) above.

5.1 Parse tree levels and bucket sort

In our implementation of the idea of constructing the canonical representation for a partial k-tree ($k \leq 3$) we are motivated by the goal of a *linear time* algorithm. Towards this goal we use the paradigm of linear time tree isomorphism algorithm, as presented in [1]. (Actually, that algorithm can be easily modified to yield a canonical representation of a tree.) We should mention two salient features of the algorithm, which we will try to emulate in the k-tree context. While labeling of the tree nodes is performed 'bottom-up', the tree is rooted and the nodes are assigned their *levels* with respect to the root, *i.e.*, distances from that distinguished node. This implies a 'top-down' preprocessing of the tree. Considering nodes by levels limits the necessity of sorting node labels to nodes of one level at a time. This, in turn, allows the bucket sort to take only linear time.

In determining the label of a hyperedge resulting from the parallel or independent rules (0.1, 0.2, 3, 4.1, 4.2, and 4.3) we need to sort the label's components. Since they are assigned numbers ranging over the interval $[1 \ldots n]$, a bucket sort guarantees only linear performance in n, the number of nodes. However, each of such (parallel or independent) reduction rule requires a sort which would force the complexity of our algorithm beyond linearity. Only when sorting label values ranging over an interval of the same cardinality as the sorted set, we get the *total* complexity of the necessary bucket sorts proportional to n.

To limit the range of label values that require sorting, we introduce the notion of a *parse tree*. For a given partial k-tree G ($k \leq 3$), each node of the unique, rooted parse tree is associated with a reduction instance. Two nodes are adjacent if the reduction instance corresponding to the child creates a hyperedge involved in the reduction corresponding to the parent node. This tree can be constructed in linear time assuming constant time per recognition of an applicable reduction rule (more about this assumption later). The root node will then be assigned level 0 and, by a top-down traversal, all other nodes can be assigned their level values.

5.2 Ready lists

The reduction instances are either bounded size, connected graphs cut off by a separator (rules 1.1, 1.2, 1.3, 4.1, 4.3, 5.1.1, 5.1.2, 5.2.3.3, 5.2.5, 6.1, 6.2, 7.1, 7.2), variable size configurations involving only vertices of degree not greater than 3 (2.1, 2.2, 5.2.3.1), parallel rules (0.1, 0.2, 3) or the buddy configuration (4.2). The membership of a vertex in such a configuration may vary during the reduction process. However, membership is altered only when one of the adjacencies of reducible vertices is altered. Since each reduction involves at most adjacencies of three remaining vertices, the total number of times a vertex changes its neighborhood or is part of a created hyperedge (by being in the separator associated with a reduction) is $\mathcal{O}(n)$, summed over all vertices. Each time this happens we investigate which is the highest ranking rule by which the vertex is reducible, or if it is a vertex of a hyperedge to which a parallel rule applies. The vertex is then linked into a list for that reduction rule.

5.3 Ring algorithm

As an exercise in algorithm design and quick and dirty programming in the Lisp-dialect *Scheme*, we include here an solution to the problem of finding the lexicographically minimum representation of a (cyclic) string of n elements $0..m(0 < 1 < \ldots < m)$. The algorithm uses few pointers to positions in the string: one pair, i_0 and j_0, in which such a minimum representation might start and another, i_1 and j_1, delimiting identical substrings. We acknowledge Alexander Proskurowski for the idea of the algorithm and Art Farley for introducing us to MacScheme.

```
(define ring
    (lambda (string m)
      (let ((i0 0) (j0 1) (i1 1) (j1 2))
        (while (and (< j0 m) (> j0 0))
           (set! j0 (remainder (+ j0 1) m))
           (while (and (and (< j0 m) (> j0 0))
                       (= (vector-ref string j0) 1))
                  (set! i1 (remainder (+ i0 1) m))
                  (set! j1 (remainder (+ j0 1) m))
                  (while (and (and (< j0 m) (> j0 0))
                              (> (vector-ref string j1) 1)
                              (= (vector-ref string i1)
                                       (vector-ref string j1)))
```

```
                    (set! i1 (remainder (+ i1 1) m))
                    (set! j1 (remainder (+ j1 1) m)))
                (cond ((= (vector-ref string i1)
                                  (vector-ref string j1))
                       (set! j0 j1))
                      ((< (vector-ref string i1)
                                  (vector-ref string j1))
                       (set! j0 (remainder (+ j1 1) m)))
                      ((> (vector-ref string i1)
                              (vector-ref string j1))
                       (set! i0 j0)
                       (set! j0 j1)))))
    i0)))
```

5.4 Data structures

An important factor in our algorithm is the ability to efficiently check adjacencies of vertices. To avoid scanning adjacency lists, we have decided to use a data structure implementing the adjacency matrix and thus requiring $\mathcal{O}(n^2)$ space. This structure, however, allows us to initialize the non-zero entries in time proportional to their number; since a partial k-tree has $\mathcal{O}(n)$ edges, this allows a linear-time preprocessing. The complexity of queries (of the type 'are vertices u and v adjacent?') is, of course, constant per query. Below follows a brief description of the structure (known from the folklore).

The initialization is performed *via* a *non-zero entry array* N, where each entry of the adjacency matrix A, say $A[u,v]$, has its unique index, say i. Thus, $N[i] = \langle (u,v), A[u,v] \rangle$. An entry of the *augmented* adjacency matrix AA, say $AA[u,v]$, has the value of $\langle i, A[u,v] \rangle$ where i is the corresponding index. At the conclusion of initialization process, a counter m contains the number of non-zero entries in A. Only when $N[\text{first}(AA[u,v])]$ is equal (u,v) and $\text{first}(AA[u,v]) \leq m$, the value of $A[u,v]$ is non-zero ('first' denotes the first element of a pair).

Similar data structures can be used to detect creation of parallel hyperedges and of buddy configurations (here, 3-dimensional matrices are used to record 3-hyperedges and adjacencies of degree 3 vertices).

The adjacency matrix does not allow us to find in constant time the neighborhood of a low degree $(1, 2$ or $3)$ vertex. Therefore we also maintain adjacency lists, in which new neighbors of vertices are added. When a vertex degree falls to 3, a *lazy delete* operation is performed, in which the whole list is scanned and reduced vertices are removed. The constant time of deleting a list element can be apportioned to the

reduction operation for the vertex, thus an amortized analysis yields linear time.

6 Conclusions

The existence of a complete set of safe reduction rules allowing recognition of partial 3-trees led us to a linear time isomorphism algorithm for this class of graphs. We expect that this can be generalized to higher values of k if such rule sets exists.

References

[1] AHO, HOPCROFT AND ULLMAN, Design and Analysis of Computer Algorithms *Addison-Wesley 1972.*

[2] S. ARNBORG, Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability — A Survey, *BIT 25 (1985), 2-33.*

[3] S. ARNBORG, D.G. CORNEIL AND A. PROSKUROWSKI, Complexity of Finding Embeddings in a k-tree, *SIAM J. Alg. and Discr. Methods 8(1987), 277-287.*

[4] S. ARNBORG AND A. PROSKUROWSKI, Characterization and Recognition of Partial 3-trees, *SIAM J. Alg. and Discr. Methods 7(1986), 305-314.*

[5] S. ARNBORG AND A. PROSKUROWSKI, Linear Time Algorithms for NP-hard Problems on Graphs Embedded in k-trees, *Discr. Appl. Math. 23(1989) 11-24.*

[6] H.L. BODLAENDER, Polynomial Algorithms for graph isomorphism and chromatic index on partial k-trees. *SWAT, Springer–Verlag LNCS 318 (1988), 227-232.*

[7] J.A. BONDY AND U.S.R. MURTY, *Graph Theory with Applications*, North Holland (1976).

[8] K.S. BOOTH, Finding a lexicographic least shift of a string, *Information Processing Letters 10* (1980), 240-242;

[9] C.J. COLBOURN AND K.S. BOOTH, Linear time automorphism algorithms for trees, interval graphs, and planar graphs, *SIAM J. Computing 10* (1981), 203-225;

[10] I.S. FILOTTI AND J.N. MAYER, A Polynomial-time Algorithm for Determining the Isomorphism of Graphs of Bounded Genus, *Proc. 12th ACM Symp. on Theory of Computing (1980), 236-243.*

[11] M. FONTET, A Linear Algorithm for Testing Isomorphism of Planar Graphs, *Proc. 3rd Int. Conf. Automata, Languages, Programming, Springer–Verlag LNCS (1976), 1411-423.*

[12] M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability*, W.H. Freeman and Company, San Francisco (1979).

[13] J.E. HOPCROFT AND J.K. WONG, A Linear Time Algorithm for Isomorphism of Planar Graphs, *Proc. 6th ACM Symp. Theory of Computer Science (1974), 172-184.*

[14] E.M. LUKS, Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time, *JCSS 25(1982), 42-65*

[15] G.L. MILLER, Isomorphism Testing for Graphs with Bounded Genus, *Proc. 12th ACM Symp. on Theory of Computing (1980), 225-235.*

[16] G.L. MILLER, Isomorphism Tesing and Cannonical Forms for k-contractible Graphs, *Proc. Foundations of Computation Theory, Springer-Verlag LNCS 158 (1983), 310-327.*

[17] A. PROSKUROWSKI, Recursive graphs, recursive labelings and shortest paths, *SIAM J. Computing 10* (1981), 391-397;

[18] A. PROSKUROWSKI, Separating Subgraphs in k-trees: Cables and Caterpillars, *Discr. Math. 49(1984), 275-285.*

[19] D.J. ROSE, Triangulated Graphs and the Elimination Process, *J. Math. Anal. Appl. 32 (1970) 597-609.*

[20] J.J. ROTMAN, *The theory of Groups (2nd ed.)*, Allyn and Bacon, 1973.

[21] P. Scheffler, Linear time algorithms for NP-complete problems for partial k-trees, *R-MATH-03/87* (1987);

[22] Y. SHILOAH, A fast equivalence-checking algorithm for circular lists, *Information Processing Letters 8* (1979), 236-238;

[23] M.M. SYSLO, Linear Time Algorithm for Coding Outerplanar Graphs, *TRN-20 1977, Institute of Computer Science, Wroclaw University.*

[24] A. WALD AND C.J. COLBOURN, Steiner Trees, Partial 2-trees, and Minimum IFI Networks, *Networks 13 (1983), 159-167.*

[25] T.V. WIMER, PhD. dissertation, Clemson University (1988);