Linear Time Isomorphism of Interval Graphs

James S. Diamond

Computer Science Department

University of Waterloo

Waterloo, Ontario.

N2L 3G1

552

ABSTRACT

A class of graphs known as interval graphs can be tested for isomorphism in O(n+e) time. This test is implemented in three stages: testing chordality to produce a list of cliques, building PQ-trees to represent the clique-vertex relationship, and testing PQ-tree isomorphism (the interval graphs are isomorphic iff the PQ-trees are isomorphic). The first stage uses an algorithm of Rose, Tarjan and Lueker; the implementation follows in this paper. The second stage is performed using an implementation by Young (as modified by Panek) of an algorithm due to Booth and Lueker. The third stage is an implementation of Colbourn and Booth's modification of Edmonds's tree isomorphism algorithm.

January 10, 1981

Table of Contents

Introduction 1

Chordality Testing 4

Interval Graph Testing 16

Isomorphism Testing 19

Conclusions 24

Acknowledgements 25

References 26

Appendix 1: Pascal Program for Chordality Testing 28

Appendix 2: Pascal Program for Isomorphism Testing 39

Appendix 3: How to Access and Run the Isomorphism Test 56

Appendix 4: Examples 58

Introduction

Although testing isomorphism for general graphs is hard (no sub-exponential algorithm is known), pairs of interval graphs can be tested in time and space linear in the size of the graphs. This testing is done by combining three algorithms. The first algorithm [Ros76] tests for chordality (a necessary condition for interval graphs) and gives a list of cliques for each graph (if the graph is chordal). The second algorithm [Boo75] attempts to build a special type of tree (called a PQ-tree) for each graph; this can be done iff the graph is an interval graph. The leaves of the PQ-tree represent cliques of the graphs; the vertex-clique relationship determines the structure of the trees. The third algorithm [Col79] tests the PQ-trees (with their leaves labelled by degree sequences) for isomorphism. This last algorithm differs from the version of Lueker and Booth [Lue79] which uses a more complicated labelling for the PQ-trees.

Given a _graph_ $G=(V,E)$, the number of _vertices_ is $n=|V|$, and the number of _edges_ is $e=|E|$. We use the usual definitions from graph theory ([Har69], [Bon76]); some additional definitions which are not standard or perhaps not well known follow.

A _chord_ of a cycle is an edge which joins two vertices not adjacent in the cycle. A graph G is _chordal_ iff every

cycle of length greater than three has a chord. Some authors use the word _triangulated_ instead of chordal.

One proper subclass of chordal graphs is the _interval graphs_. Interval graphs are those graphs that can be represented as an intersection model of intervals on the real line. Given a set of intervals, associate one vertex with each interval; two vertices are adjacent iff their corresponding intervals have a non-empty intersection.

A _clique_ is a _maximal_ complete subgraph. This definition differs from that of some authors, who don't require maximality.

A matrix of zeroes and ones is said to have the _consecutive ones_ property if there exists a permutation of the rows so that in the permuted matrix the ones in each column occur consecutively. The following characterization of interval graphs is proved by Fulkerson and Gross [Ful65]. A graph G is an interval graph iff the clique vs. vertex incidence matrix of G has the consecutive ones property. It is not necessary to explicitly construct the matrix. Rather, it suffices to find an ordering of the cliques so that the set of cliques for each vertex is contiguous within that ordering.

A _PQ-tree_ is a tree with two types of interior nodes: P nodes, whose children may be arbitrarily permuted, and Q

nodes, whose children may only be reversed left for right. For a complete definition of PQ-trees, see [Boo75].

The following section discusses the first of the three algorithms: the chordality test. Testing for the interval graph property, which uses the PQ-tree algorithms, is then mentioned briefly. The final main section deals with the isomorphism test.

Chordality Testing

Chordality testing is performed in two stages. In the first, the ordering algorithm assigns a number and a label to each vertex. Labels consist of a string of vertex numbers; the exact construction of this string follows below. The second stage arises from the following observation (which is a consequence of the way in which labels are assigned): the vertices in each label form a complete subgraph iff the graph is chordal. Maximal labels correspond to cliques; these are important for the interval graph test, so we will want to distinguish maximal labels from non-maximal ones.

The labelling is done with a *lexicographic breadth-first search*. The search is breadth-first, as each neighbour w of v is examined when v is processed. The search is lexicographic, as the queue of vertices yet to be examined is ordered by the lexicographic rank of the labels. If appropriate care is taken in the labelling procedure, it can be accomplished in O(n+e) time. The lexicographic ordering is the usual dictionary ordering, except that integers are used in lieu of letters of the alphabet.

As well as assigning labels to vertices, the algorithm assigns a *number* to each vertex. This number is not to be confused with the *name* of the vertex, which is a necessary means of identification.

Initially, each vertex is unnumbered and has a null label. Note that the null label is considered to be lexicographically less than any non-null label. The vertices are numbered and labelled in an iterative process. At step k (for k from n down to 1), an unnumbered vertex v with maximal label is chosen. Vertex v then receives the number k. All vertices adjacent to v as well as v itself then have k appended to their label. This step is repeated n times, after which the graph has been lexicographically ordered. The basic algorithm is shown in Figure 1. Figure 2 shows a step-by-step labelling of a graph. The label of each vertex is a sequence of integers enclosed in parentheses; a right parenthesis signifies a completed label and hence a numbered vertex. The number of a vertex is always the last integer of its completed label.

```
Mark each vertex unnumbered;
Give each vertex a null label;
FOR i := n TO 1 STEP -1 DO
BEGIN
  Choose an unnumbered vertex v with
     lexicographically maximum label;
  Give v the number i;
  Append i to v's label;
  FOR all unnumbered vertices w adjacent to v DO
    append i to w's label;
END
```

Figure 1.

The basic labelling algorithm.

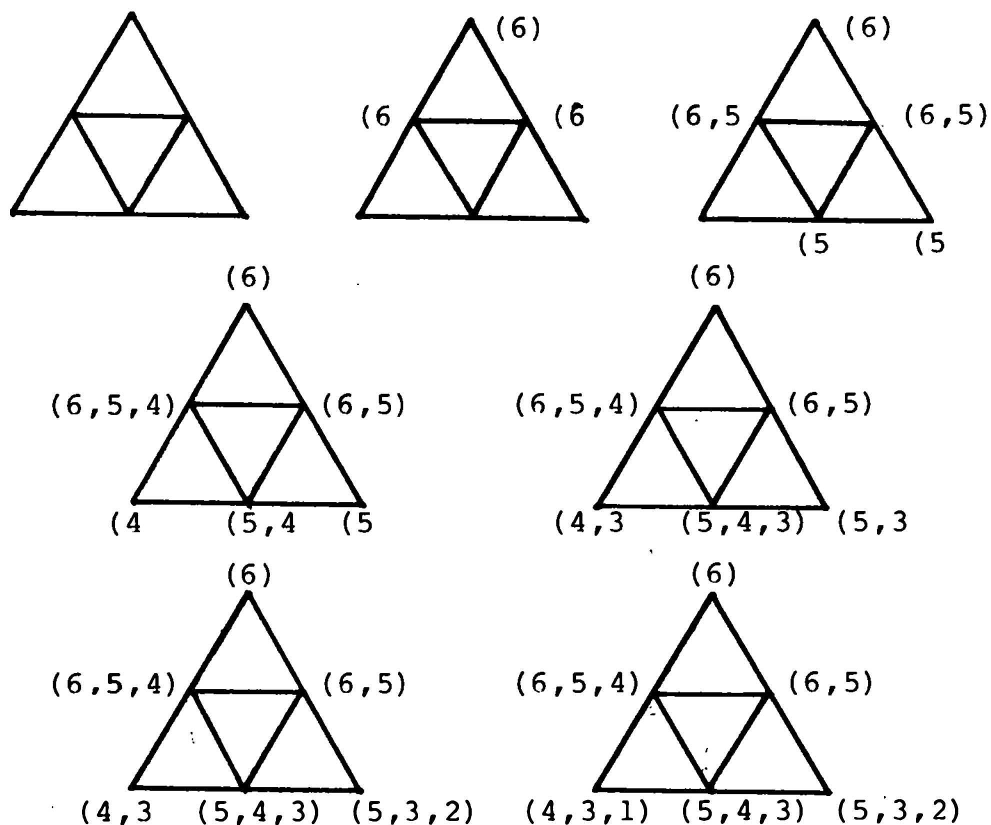


Figure 2.

An example of the labelling algorithm.

The lexicographic breadth-first search algorithm is interesting for the reason mentioned above: the vertices in each label form a complete subgraph iff the graph is chordal. To prove this, we need the following:

<u>Lemma</u>: Let G be any graph numbered by lexicographic breadth-first search. If j and k are vertices whose numbers are larger than i and both are adjacent to i then there is a path between j and k, all of whose

internal vertices are both nonadjacent to i and numbered larger than i.

Proof: the proof is found in [Boo75] and will not be repeated here.

The lemma allows us to prove the following:

Theorem: A graph G is chordal iff the lexicographic breadth-first search labels form complete subgraphs.

Proof: This proof is a modification of one found in [Boo75].

Proof =>: Suppose that G is chordal. Number the vertices according to lexicographic breadth-first search. The above lemma can be restated in a stronger form.

Claim: If $i < j < k$ and j and k are both adjacent to i, then they are also adjacent to each other.

Proof of claim: Using the lemma, there must be a path, nonadjacent to i, whose internal vertices connect i and j and are all numbered higher than i. Choose a minimal path which satisfies this condition. Add the edges $\{i,j\}$ and $\{i,k\}$ to form a cycle. The cycle obviously has no chord, because the path is minimal and i is only adjacent to j and k, by assumption. But if the cycle has length greater than three, the graph could not be chordal. This implies that the path in question

actually is a single edge and hence that j and k are adjacent in G.

Now consider the label of vertex i. By inspection of the lexicographic breadth-first search algorithm, we see that for every pair of vertices $j,k > i$ in the label, both j and k must be adjacent to i. Without loss of generality, assume $j < k$. Then the claim tells us that j and k are adjacent. Thus the label forms a complete subgraph.

Proof <=: Suppose that the labels for G given by the lexicographic breadth-first search form complete subgraphs of G. Consider any cycle of length greater than three. Select i to be the lowest numbered vertex in the cycle and let j and k be the two immediately adjacent vertices in the cycle. As j and k are adjacent to i, and $j,k > i$, we must have both j and k in i's label. By the supposition, j is adjacent to k. Hence G is chordal.

Performing the labelling in linear time is really just an exercise in list handling. Labels are not explicitly stored with each vertex. Rather, we have a queue of lists, where each list represents all vertices which (currently) have the same label. The queue is arranged so that the lists are lexicographically ordered; the vertices in the first list have the lexicographically maximal label, the vertices in the second list have the second largest label,

etc. This scheme provides the implicit labelling. The data structure is not a queue in the strict sense of the word: as seen below, insertions into the middle of the queue are needed. This forces the queue to be implemented as a doubly linked list.

Aside from the forward and backward links (respectively called 'head' and 'back') each queue element (called a 'cell') has two additional fields. 'Next' is a pointer to the first element in the list, and 'flag' indicates whether or not this list is 'new' (see below). The elements in the lists use basically the same data structure. 'Next' and 'back' are used to doubly link all vertices (i.e. all cells corresponding to vertices) with the same label. 'Flag' points to the queue entry for this list, and 'head' points to the vertex corresponding to this cell. (In the Pascal implementation these cells have the 'head' field called 'VertName' for clarity.)

Initially, there is only one list on the queue; this list contains all of the vertices, since all vertices start with the null label, and the list is marked 'old'. At each step, a member v of the first non-empty list is chosen, removed from the list, and numbered. Now comes the tricky part! We would like to append the current number to the label of all unnumbered vertices w adjacent to v. This requires creating zero or more new lists (zero if v is not adjacent to any unnumbered vertices). Further, if two

vertices w and x are both adjacent to v, and are both currently on the same list, (i.e. they have the same label), after the list structure is modified w and x must again be on the same list. This can be done as follows.

As each w adjacent to v is found, remove w from its list (say l). Look at the list (lexicographically) above l in the queue (if any). If such a list exists and is marked 'new', add w to this list. Otherwise, create a new list, mark it 'new', and add w to this list. Insert this list directly above l in the queue. This scheme insures that w and x are on the same list at the end of the step.

After v's adjacency list has been entirely examined, mark all 'new' lists 'old'. This can be done without examining all lists by keeping a list of all 'new' lists; whenever a new list is created, make a pointer to it and store this pointer in another list. Figures 3 and 4 show the linked list structure after vertices 5 and 4 of the sample graph have been labelled. For simplicity, the vertices of the graph have names equal to their lexicographic breadth-first search numbers.

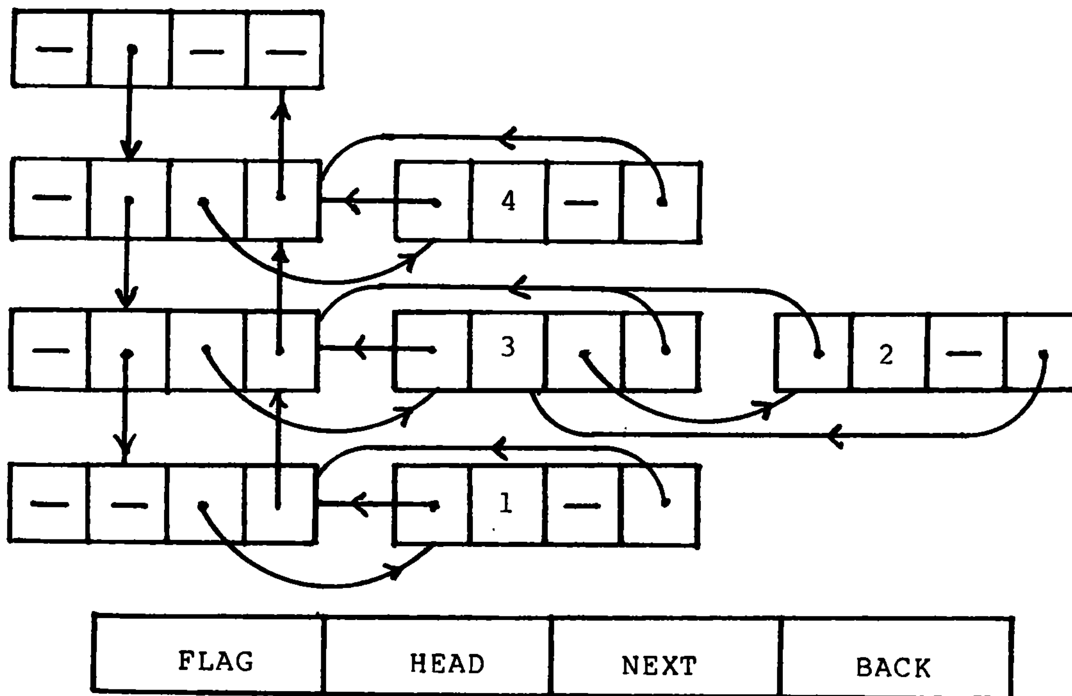


Figure 3.

The linked list structure after vertex 5 is removed.

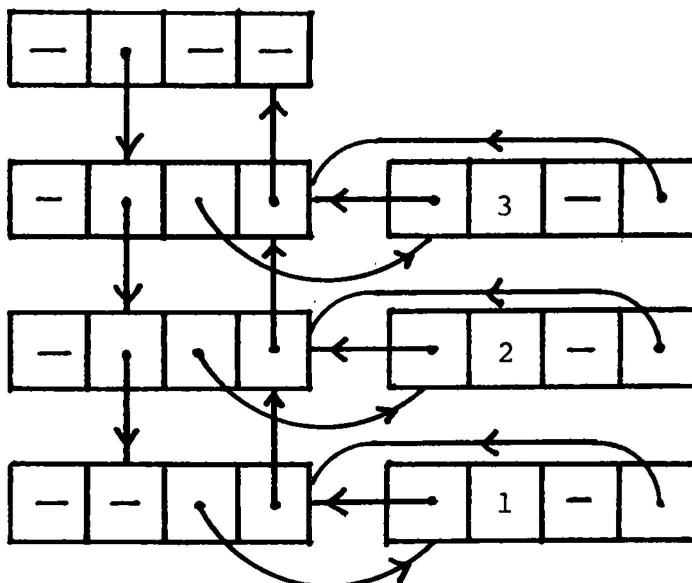


Figure 4.

The linked list structure after vertex 4 is removed.

A detailed algorithm for this process follows. Note that this algorithm is slightly different from the one given by Rose, Tarjan and Lueker [Ros76] because of some apparent errors detected in their algorithm. On line 19 of the algorithm in [Ros76] the first ':=' should be '='. Lines 24 and 25 should be replaced with 'if next(p) <> 0 then back(next(p)) := back(p))'. Finally, a more subtle error occurs on line 50 where '...flag(h)=v' should read '...flag(h)=0'. A corrected version of the algorithm follows in Figure 5.

```
BEGIN
    { (implicitly) assign null label  }
    to all vertices;
  head(1) := 2;
  back(2) := 1;
  head(2) := back(1) := next(1) := flag(1)
          := flag(2) := 0;
  c := 3;
    { c is the number of the first empty cell;  }
  FOR all vertices v DO
  BEGIN
    head(c) := v;
    cell(v) := next(c-1) := c;
    flag(c) := 2;
    back(c) := c - 1;
    c := c + 1;
    number(v) := 0;
  END;
  next(c-1) := 0;
  FOR i := number of vertices to 1 STEP -1 DO
  BEGIN
      { skip empty sets;  }
    WHILE next(head(1)) = 0 DO
    BEGIN
      head(1) := head(head(1));
      back(head(1)) := 1;
    END;
      { pick next vertex to number;  }
select: p := next(head(1));
      { delete p's cell from its set;  }
    next(head(1)) := next(p);
    IF next(p) <> 0
      THEN back(next(p)) := back(p);
```

```
        { assign v the number i;  }
      name_of(i) := v;
      number(v) := i;
      fixlist := null;
  update: FOR all w adjacent to v DO
        IF number(w) = 0 THEN
      BEGIN
          { delete cell of w from its set;  }
        next(back(cell(w))) := next(cell(w));
        IF next(cell(w)) <> 0 THEN
          back(next(cell(w))) := back(cell(w));
        h := back(flag(cell(w)));
          { if h is an old set then create a new set;  }
        IF flag(h) = 0 THEN
        BEGIN
          head(c) := head(h);
          head(h) := c;
          back(head(c)) := c;
          back(c) := h;
          flag(c) := 1;
          next(c) := 0;
          add c to fixlist;
          h := c;
          c := c + 1;
        END;
          { add cell of w to this new set;  }
        next(cell(w)) := next(h);
        IF next(h) <> 0 THEN back(next(h)) := cell(w);
        flag(cell(w)) := back(cell(w)) := h;
        next(h) := cell(w);
      END;
      FOR all h in fixlist DO flag(h) := 0;
    END;
  END
```

Figure 5.

Detailed labelling algorithm.

There are two points left to be determined:

(1) how are the labels actually produced, and

(2) how are the <u>maximal</u> labels (efficiently) found?

To produce the labels, two additional fields were added to the 'cell' structure given in the algorithm of [Ros76]. One field, 'LabelPtr', is a pointer to a cell. The second,

'LabelNum', is an integer. Whenever a new list is created, LabelNum of the list header is set to the number of v (the vertex just numbered). This indicates that all vertices which will be on this set have this number as part of their label. As they are yet unnumbered, and will be put on this list iff they are adjacent to v, they receive v's number in their label exactly as they should. At the same time, LabelPtr of the new list header is made to point to the list (currently) directly below it in the queue; this is l in the description above Figure 3. By chaining through these pointers (the initial list in the queue has a null LabelPtr as it has a null label) the complete label for any vertex can be found. The initial list was not created by numbering a vertex and thus has no number.

When a vertex is numbered, it is easy to determine whether or not its label is maximal. As stated above, the vertex v to be numbered is chosen from the list having the lexicographically greatest label. If no vertex on this list is adjacent to v, then v's label <u>can not</u> be the prefix of any other label; if a vertex on this list is adjacent, then v's label <u>will</u> be the prefix of another label. Checking each w adjacent to v for this condition is easily done: the algorithm supplies each cell with a pointer to its list header (i.e. 'flag').

The final stage of the chordality testing requires checking the labels to insure that they form complete

subgraphs. We don't want to check all pairs of each label: as we have n labels, and each has O(n) elements, examining all pairs would require $O(n^3)$ work, violating the O(n+e) time bound. We first create a list of pairs (i,j) where i<j and vertex i is adjacent to vertex j; this is a list of actual edges. Next we create the list of necessary edges. For each label of length greater than two, form all pairs of the form (i,j) where i is the <u>second</u> smallest element of the label and the j's are all integers in the label greater than i.

It is sufficient to check only these pairs for two reasons: (1) with the described method of assigning labels, the smallest element must be adjacent to everything else in the label, and (2) the pairs comprised of elements greater than the second smallest are checked while processing labels of other vertices.

The two lists of edges are then bucketsorted; the bucketsorting of O(e) pairs over a range of n takes at most O(n+e) time. One pass through the lists suffices to insure that the needed edges form a subset of the actual edges. Examining the theorem, we see that if the needed edges are a subset of the edges then the graph is chordal.

Interval Graph Testing

As mentioned earlier, a chordal graph is an interval graph iff there exists an ordering of its cliques so that the set of cliques containing a vertex v is contiguous within that order, for all vertices v.

The output of the chordality test gives us the cliques (if the graph is chordal). Using the PQ-tree data structure, in O(n+e) time it can be decided whether or not a chordal graph is an interval graph ([Lue79]). The algorithm is quite complicated and will not be discussed here.

Using Panek's [Pan78] modification of Young's [You77] implementation, PQ-trees corresponding to interval graphs can be built. If the chordal graph is not an interval graph, the program stops and reports this. If the graph is indeed an interval graph, an encoding of the PQ-tree is output.

As an example, see the following figures. Figure 6 shows a three component graph, and Figure 7 lists the cliques for this graph. Figure 8 shows the corresponding PQ-tree as determined by the program and the output given by Panek's implementation with letters substituted for digits. The P nodes are drawn as circles, and the Q nodes are drawn as rectangles. The program output represents this with angle brackets ('<>') and square brackets ('[]') respectively.

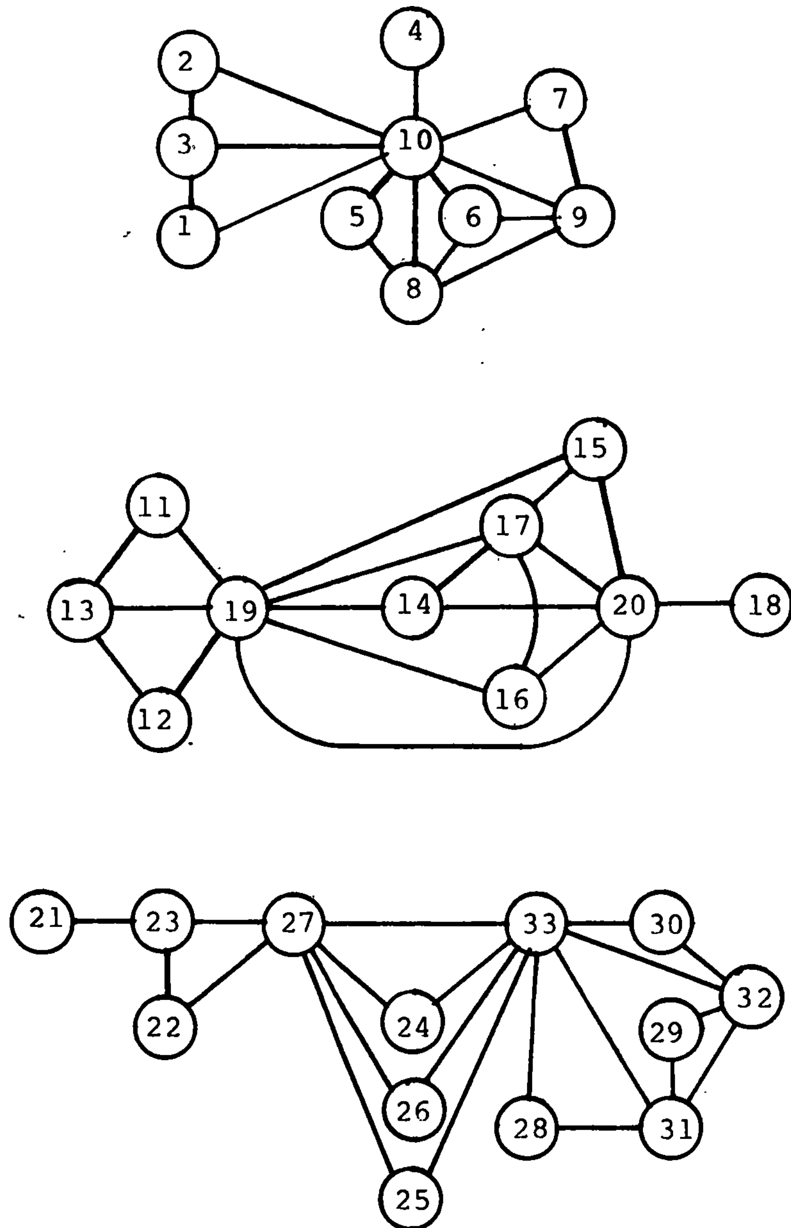


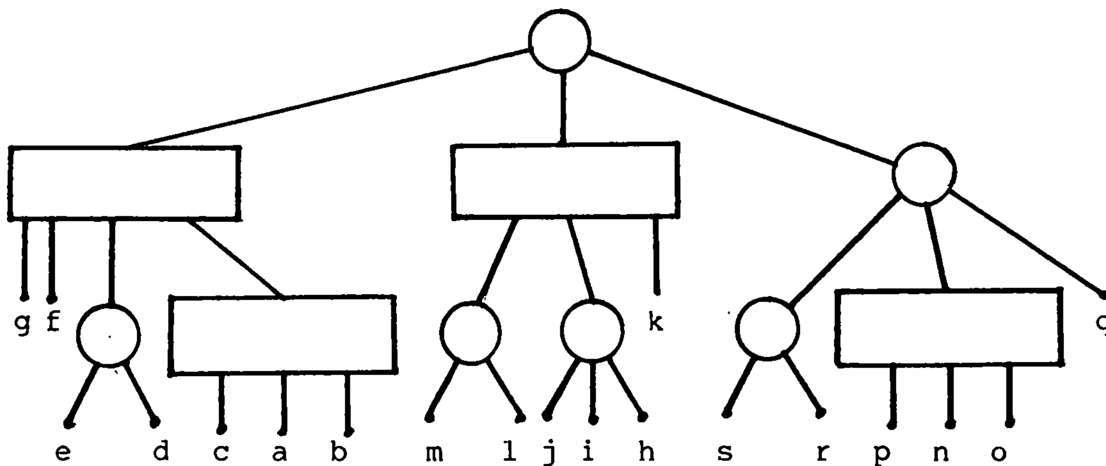
Figure 6.

A three component interval graph.

Clique		Vertices
1	a	29 31 32 33
2	b	30 32 33
3	c	28 31 33
4	d	25 26 27 33
5	e	24 26 27 33
6	f	22 23 27
7	g	21 23
8	h	16 17 19 20
9	i	15 17 19 20
10	j	14 17 19 20
11	k	18 20
12	l	12 13 19
13	m	11 13 19
14	n	6 8 9 10
15	o	7 9 10
16	p	5 8 10
17	q	4 10
18	r	2 3 10
19	s	1 3 10

Figure 7.

The cliques of the graph of figure 6.



<[g f <e d>[c a b]][<m l><j i h> k]<<s r>[p n o] q>>

Figure 8.

The PQ-tree for the graph of Figure 6.

Isomorphism Testing

Isomorphism testing for interval graphs) is almost as easy as isomorphism testing of the PQ-trees given by the interval graph test. Testing the trees with no additional information does not suffice, as a PQ-tree does not by itself give a canonical representation of an interval graph. For example, in Figure 9 two different interval graphs (modified from graphs in [Lue79]) are shown which have the same PQ-tree, shown in Figure 10. However, as shown in [Col80], labelling the leaves (cliques) of the PQ-trees with the degree sequences does give a canonical representation up to isomorphism.

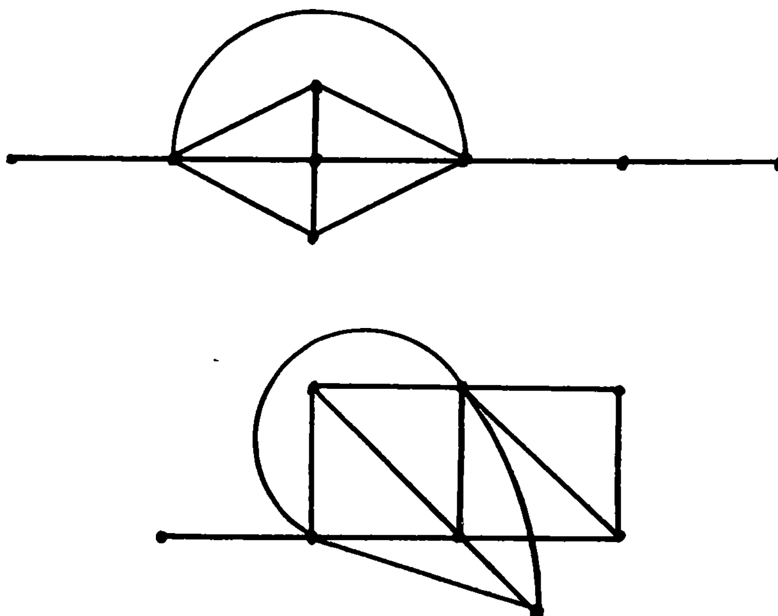


Figure 9.

Two non-isomorphic graphs with the same PQ-tree.

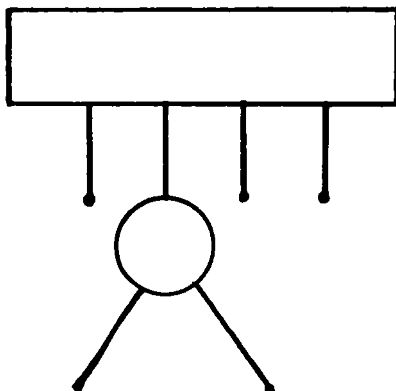


Figure 10.

The PQ-tree of the graphs in Figure 9.

The first part of the testing is to produce the sorted degree sequences for the cliques. With a bucket sort, this can be done in O(n+e) time. Each leaf is given an <u>i-number</u> which is equal to the rank of its degree sequence within the sorted list of degree sequences.

Isomorphism testing proceeds as follows. At every level of the forest, beginning with the leaves of maximum depth, the <u>labels</u> of all nodes at that depth are sorted. The label of a leaf is its i-number; the label of a P node is '1' followed by the sorted sequence of its children's i-numbers; the label of a Q node is '2' followed by the i-numbers of its children: left to right, or right to left, whichever is lexicographically smaller. Then each node's i-number is set equal to the rank of its label.

The above steps are repeated until the roots have received their i-numbers. The trees are isomorphic iff their roots have the same i-numbers. See Figure 11 for the pseudo code.

```
BEGIN
  Sort the clique degree sequences;
  Assign each leaf's i-number as the rank of
    its degree sequence;
  FOR d := maxi…um depth TO 1 STEP -1 DO
  BEGIN
    { Create labels at this level }
    FOR each node at this level DO
    IF node if a leaf THEN label := i-number
      ELSE IF node is a P node
             THEN label := 1, <sorted sequence
                              of children's i-numbers>
             else label := 2, min(<children's i-numbers
                             left to right>, <children's
                              i-numbers right to left>);
    Sort all labels at this level;
    FOR each node at this level DO
      i-number := rank of label;
  END;
  IF i-numbers of both roots are equal
    THEN isomorphic := TRUE
    ELSE isomorphic := FALSE;
END
```

Figure 11.

PQ-tree isomorphism algorithm.

There are only two details of this algorithm which bear comment. The first is the method of accessing nodes at a particular level in the forest. The second is sorting variable length strings of integers.

Searching through the forest at every iteration for nodes of a particular depth is too inefficient. However, with a depth first search of each tree, it is easy to create

a list of nodes for each level. This allows us to build these lists in time proportional to the size of the forest.

To sort variable length strings, a slight modification of an algorithm in [Aho74] is used. The modification, done for ease of programming, 'sorts' the strings in an order different than the usual lexicographic order. This modification does not increase the linear time bound of the algorithm. As it turns out, this modification of the ordering is not important; the only necessary requirement of the 'sort' is that any set of equal strings be sorted into one contiguous sequence.

As shown in [Aho74], the labels of a forest (two trees, in our case) can be sorted in time proportional to the size of the forest plus the sum of the lengths of the original labels. The original labels are the clique degree sequences; thus the sum of their lengths is $O(n+e)$. Also, determining isomorphism of two n-vertex trees with labels in the range 1 to n can be done in $O(n)$ time. Hence the isomorphism testing can be performed in $O(n+e)$ time.

A more detailed explanation of the algorithm is found in Figure 12.

```
BEGIN
  sort clique degree sequences;
  FOR each leaf v in the forest DO
    v's i-number := rank of corresponding sequence;

  FOR l := maximum depth in forest TO 1 STEP -1 DO
  BEGIN
    initialize label list to null;
    FOR every node v at depth l DO
    BEGIN
      { create v's label }
      IF v is a leaf THEN v's label := v's i-number
      ELSE IF v is a P node THEN
        BEGIN
          get i-numbers of v's children;
          sort them in ascending order;
          v's label := this sorted list;
        END
        ELSE
        BEGIN
          { v is a Q node }
          get i-numbers of v's children left to right;
          get i-numbers of v's children right to left;
          v's label := lexicographic minimum of
                          the above sequences;
        END;
      add v's label to the list of labels;
    END;
    sort the list of labels;
    FOR every node v at depth l DO
      v's i-number := v's label's rank in sorted list;
  END;

  IF both roots have same i-number
    THEN isomorphic := TRUE
    ELSE isomorphic := FALSE:
END
```

Figure 12.

Isomorphism testing algorithm.

Conclusions

The chordality test and isomorphism test algorithms were implemented in Pascal on the University of Waterloo Math Faculty Computing Facility Honeywell 66/60. Panek's implementation of the interval graph test is also implemented in Pascal on the same computer.

The programs were then tested on various graphs. Some of these appear in Appendix 4.

As a final note, a relatively small extension of the isomorphism test will give the automorphism partition of the forest. Most of the necessary work is in sorting variable length labels, and this is already done for the isomorphism test. The only non-trivial part remaining is to find the node (or part of node!) of the PQ-tree which corresponds to each vertex of the original graph. This is described in detail in [Col80].

Acknowledgements

It seems fitting at this point to quote Dolores Panek, a former student of Kelly Booth:

> "The writer is grateful to Kelly Booth for patience far beyond any expectation."

January 10, 1981

References

[Aho74] - Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. 1974. The Design and Analysis of Computer Algorithms. Reading, Massachusetts: Addison-Wesley.

[Bon76] - J.A. Bondy and U.S.R. Murty. 1976. Graph Theory with Applications. New York: North Holland.

[Boo75] - Kellogg S. Booth. November 1975. PQ-Tree Algorithms. Ph.D. Thesis, University of California at Berkeley.

[Col80] - Charles J. Colbourn and Kellogg S. Booth. March, 1980. Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs. Dept. of Computer Science Report CS-79-06, University of Waterloo.

[Ful65] - D.R. Fulkerson and O.A. Gross. 1965. "Incidence Matrices and Interval Graphs," Pacific Journal of Mathematics, Vol. 15, No. 3.

[Lue79] - George S. Lueker and Kellogg S. Booth. April 1979. "A Linear Time Algorithm for Deciding Interval Graph Isomorphism," Journal of the Association for Computing Machinery, Vol. 26, No. 2, pp. 183-195.

[Har69] - Frank Harary. 1969. _Graph Theory_. Reading, Massachusetts: Addison-Wesley.

[Pan78] - Dolores M. Panek. 1978. _Implementing a Linear-Time Test for Graph Planarity_. Master's Essay, Dept. of Computer Science, University of Waterloo.

[Ros76] - Donald J. Rose, R. Endre Tarjan and George S. Lueker. June 1976. "Algorithmic Aspects of Vertex Elimination on Graphs," _SIAM Journal on Computing_, Vol. 5 No. 2.

[You77] - Sarah M. Young. 1977. _Implementation of PQ-Tree Algorithms_. Master's Thesis, Dept. of Computer Science, University of Washington.

Appendix 1

A Pascal Program for Chordality Testing

The following routines implement the chordality test. They are found under the catalog gr/./zsd/iso-test/chordal.

gr/./zsd/iso-test/chordal/types.p:

```
const   MaxVerts = 50;

type    EPointer = ^edge;

        CPointer = ^CellNode;

        CliqLPtr = ^CliqueList;

        edge = record
        {begin}
           next:                      EPointer;
           OtherEnd:                  integer;
        end;

        vertex = record
        {begin}
           AdjList:                   EPointer;
           LexNum:                    integer;
           degree:                    integer;
           cliques:                   CliqLPtr;
           cell:                      CPointer;
        end;

        graph = record
        {begin}
           NumVerts:                  integer;
           NumCliques:                integer;
           vertices:                  array[1..MaxVerts] of vertex;
           NameOf:                    array[1..MaxVerts] of integer;
        end;

        CellNode = record
        {begin}
           next:                      CPointer;
           back:                      CPointer;
           flag:                      CPointer;
           LabelPtr:                  CPointer;
           LabelNum:                  integer;
           case boolean of
                true:  (head:           CPointer);
```

```
                false: (VertName:  integer);
        end;

        CliqueList = record
        {begin}
           next:                   CliqLPtr;
           CliqNum:                integer;
        end;

        text = file of char;
```

gr/./zsd/iso-test/chordal/input.p:

```
{ This procedure reads in the graph from the tty or a file.
  The first line should be the number of vertices, and the
  subsequent lines should be the adj. lists for each vertex.
  Each adjacency list should have the end flagged with a 0. }

procedure InputGraph(var g: graph);
var     i, num:                 integer;
        ej:                     EPointer;
        Fromtty:                boolean;
        last:                   EPointer;
        filename:               packed array [1..40] of char;
        infile:                 file of char;

begin
   writeln('If data in a file, type name; else return');
   readln(filename);
   Fromtty := filename[1] = ' ';
   if not Fromtty then openf(infile, filename, 'r')
                  else openf(infile, '  ', 'r');
   with g do
   begin
      if Fromtty then
         write('How many vertices (max = ',MaxVerts:1,') ? ');
      readln(infile, NumVerts);

      if Fromtty then
         write('List adjacent vertices for each vertex;');
         writeln(' end with a 0');
      for i := 1 to NumVerts do
      begin
         new(vertices[i].AdjList);
         vertices[i].AdjList^.next := nil;
         last := vertices[i].AdjList;
         vertices[i].degree := 0;
         vertices[i].cliques := nil;
         if Fromtty then write(i:1, ': ');
```

```
         read(infile, num);
         while num <> 0 do
         begin
            vertices[i].degree := vertices[i].degree + 1;
            new(ej);
            last^.OtherEnd := num;
            last^.next := ej;
            ej^.next := nil;
            last := ej;
         read(infile, num);
         end; {of while}
      end; {of for}
   end; {of with g}
   if not Fromtty then closef(infile);
   writeln;
   writeln(' *** input complete ***');
   writeln;
end; {of InputGraph}
```

gr/./zsd/iso-test/chordal/lex-order.p:

```
{ This procedure does the lexicographic breadth first
  search as specified by Rose, Tarjan and Lueker.
  It also writes out the vertex labels as they are found.
  The maximal labels are numbered, and the non-maximal labels
  are flagged with a '*'.  }

procedure LexOrder(var g: graph; var outfile: text);
type    LPointer = ^ListEl;

        ListEl = record
        {begin}
           next:                           LPointer;
           cell:                           CPointer;
        end;

var     w:                                 EPointer;
        FixList:                           LPointer;
        ListElm:                           LPointer;
        WCell:                             CPointer;
        first:                             CPointer;
        second:                            CPointer;
        empty:                             CPointer;
        old:                               CPointer;
        h:                                 CPointer;
        temp:                              CPointer;
        v:                                 CPointer;
        i:                                 integer;
        maximal:                           boolean;
```

```
procedure AddToCliqueList(CliqueNumber, VertexName: integer);
var     CLP:                              CliqLPtr;
begin
   new(CLP);
   CLP^.CliqNum := CliqueNumber;
   CLP^.next := g.vertices[VertexName].cliques;
   g.vertices[VertexName].cliques := CLP;
end;

begin
   { Write out the headings for the label lists }
   writeln(outfile,
      'The labels generated by the lexicographic BFS are:');
   writeln(outfile,
      'Vertex names                          BFS numbers');

   { Initialize }
   new(FixList);
   new(first);
   new(second);
   first^.head := second;
   second^.back := first;
   first^.back := nil;
   first^.flag := nil;
   first^.next := nil;
   second^.flag := nil;
   second^.head := nil;
   second^.LabelPtr := nil;
   g.NumCliques := 0;

   { Empty is (a pointer to) the next cell to be used }
   new(empty);

   { Initialize the list of vertices }
   old := second;
   for i := g.NumVerts downto 1 do
   begin
      empty^.VertName := i;
      g.vertices[i].cell := empty;
      old^.next := empty;
      empty^.flag := second;
      empty^.back := old;
      old := empty;
      new(empty);
      g.vertices[i].LexNum := 0;
   end;
   old^.next := nil;

   { Do the lexicographic ordering }
   for i := g.NumVerts downto 1 do
   begin
      { Initialize this iteration }
      maximal := true;
      FixList^.next := nil;
```

```
      { Skip empty sets }
      while first^.head^.next = nil do
      begin
         first^.head := first^.head^.head;
         first^.head^.back := first;
      end;

      { Get a vertex with maximum label }
{select:} v := first^.head^.next;
      { Delete it from it's set }
      first^.head^.next := v^.next;
      if v^.next ~= nil then v^.next^.back := v^.back;

      { Give the vertex the number i }
      g.NameOf[i] := v^.VertName;
      g.vertices[v^.VertName].LexNum := i;

{update:} w := g.vertices[v^.VertName].AdjList;
      while w^.next ~= nil do
      begin
         WCell := g.vertices[w^.OtherEnd].cell;
         { Is this vertex yet to be lexicographically numbered? }
         if g.vertices[WCell^.VertName].LexNum = 0 then
         begin
            { If w was in same set as v then
              the label is not maximal }
            if WCell^.flag = v^.flag then maximal := false;

            { Delete cell of w from it's set }
            WCell^.back^.next := WCell^.next;
            if WCell^.next ~= nil
               then WCell^.next^.back := WCell^.back;
            h := WCell^.flag^.back;

            { If h is an old set then create a new set }
            if h^.flag = nil then
            begin
               empty^.head := h^.head;
               h^.head := empty;
               empty^.head^.back := empty;
               empty^.back := h;
               empty^.flag := first;
               empty^.next := nil;
               empty^.LabelPtr := WCell^.flag;
               empty^.LabelNum := i;

               { Add empty to FixList }
               new(ListElm);
               ListElm^.cell := empty;
               ListElm^.next := FixList^.next;
               FixList^.next := ListElm;

               h := empty;
               new(empty);
```

```
        end;

        { Add cell of w to new set }
        WCell^.next := h^.next;
        if h^.next ~= nil then h^.next^.back := WCell;
        WCell^.flag := h;
        WCell^.back := h;
        h^.next := WCell;
    end; { of if LexNum = 0 }

    w := w^.next;
end; { of while w ~= nil }

{ Fix the flags - make the new sets old }
ListElm := FixList^.next;
while ListElm ~= nil do
begin
    ListElm^.cell^.flag := nil;
    ListElm := ListElm^.next;
end;

{ Output the label - first a number iff maximal, then
  the label with vertex names, then the label with
  BFS numbers. }
if maximal then
begin
    g.NumCliques := g.NumCliques + 1;
    write(outfile, ' ',g.NumCliques:2, '   ', g.NameOf[i]:1);
    AddToCliqueList(g.NumCliques, g.NameOf[i]);
end
else write(outfile, '*     ', g.NameOf[i]:1);

{ The label with original vertex names... }
temp := v^.flag;
while temp^.LabelPtr ~= nil do
begin
    write(outfile, ',', g.NameOf[temp^.LabelNum]:1);
    { For each vertex in the clique, add this clique
      number to it's list of cliques }
    if maximal then AddToCliqueList(g.NumCliques,
                              g.NameOf[temp^.LabelNum]);

    temp := temp^.LabelPtr;
end;

{ The label with the lexicographix BFS numbers... }
temp := v^.flag;
write(outfile, '                    ', i:1);
while temp^.LabelPtr ~= nil do
begin
    write(outfile, ',', temp^.LabelNum:1);
    temp := temp^.LabelPtr;
end;
writeln(outfile);
```

```
   end; { of for statement }
end; { of LexOrder }
```

gr/./zsd/iso-test/chordal/output.p:

```
procedure OutputGraph(g: graph; var outfile: text);
var     i:                                  integer;
        DegreeSum:                          integer;
        NumEdges:                           integer;
        ej:                                 EPointer;
        CLP:                                CliqLPtr;
begin
   with g do
   begin
      DegreeSum := 0;
      writeln(outfile);
      writeln(outfile, 'Name: LexNum: ',
         'Degree: AdjList:   Clique Membership List');
      for i := 1 to NumVerts do
      begin
         DegreeSum := DegreeSum + vertices[i].degree;
         write(outfile, i:3, ':  ');
         write(outfile, vertices[i].LexNum:3, ':     ');
         write(outfile, vertices[i].degree:3, ':     ');
         ej := vertices[i].AdjList;
         while ej^.next ~= nil do
         begin
            write(outfile, ej^.OtherEnd:1, ' ');
            ej := ej^.next;
         end;
         write(outfile, ':             ');
         CLP := vertices[i].cliques;
         while CLP ~= nil do
         begin
            write(outfile, CLP^.CliqNum:1, ' ');
            CLP := CLP^.next;
         end;
         writeln(outfile);
      end; {of for}
      writeln(outfile);
      writeln(outfile, ' Vertices     Edges     Cliques');
      NumEdges := DegreeSum div 2;
      writeln(outfile, NumVerts:7, NumEdges:11, NumCliques:12);
   end; {of with}
end; {of OutputGraph}
```

gr/./zsd/iso-test/chordal/c-test.p:

```
{ This procedure creates a list of the necessary edges
  (necessary for chordalaity) and a list of all edges.
  It then sorts each list (using a bucket sort) and
  compares the lists to insure that the list of
  necessary edges is a subset of the list of actual
  edges.  }

function CheckChordality(g: graph):  boolean;
type    ElPtr = ^ListEl;
        ListEl = record
        {begin}
           v:                     array[1..2] of integer;
           pred:                  ElPtr;
           suc:                   ElPtr;
        end;

var     tempc:                    CPointer;
        tempe:                    EPointer;
        act:                      ElPtr;
        actual:                   ElPtr;
        need:                     ElPtr;
        needed:                   ElPtr;
        i:                        integer;
        LexOfI:                   integer;
        second:                   integer;

procedure insert(list: ElPtr; v1, v2: integer);

var     ListElm:                  ElPtr;

begin
   new(ListElm);
   ListElm^.v[1] := v1;
   ListElm^.v[2] := v2;
   ListElm^.pred := list^.pred;
   ListElm^.pred^.suc := ListElm;
   list^.pred := ListElm;
end; { of procedure insert }

procedure BucketSort(list: ElPtr);

var     bucket:                   array[1..MaxVerts] of ElPtr;
        temp:                     ElPtr;
        temp2:                    ElPtr;
        i, j:                     integer;

procedure move(item, ThisBucket: ElPtr);
begin
   item^.pred := ThisBucket^.pred;
   item^.pred^.suc := item;
   ThisBucket^.pred := item;
end;
```

```
begin
   { Initialize the buckets }
   for i := 1 to g.NumVerts do new(bucket[i]);

   for i := 2 downto 1 do
   begin
      for j := 1 to g.NumVerts
                 do bucket[j]^.pred := bucket[j];
      temp := list^.suc;
      while temp ~= nil do
      begin
         temp2 := temp^.suc;
         move(temp, bucket[temp^.v[i]]);
         temp := temp2;
      end;

      { Concatenate the buckets back into 1 list }
      list^.pred := list;
      for j := 1 to g.NumVerts do
         if bucket[j]^.pred ~= bucket[j] then  { if ~ empty }
      begin
         list^.pred^.suc := bucket[j]^.suc;
         bucket[j]^.suc^.pred := list^.pred;
         list^.pred := bucket[j]^.pred;
      end;
      list^.pred^.suc := nil;
   end; { of for i }
end; { of procedure BucketSort }

{ Finally....begin the function body! }
begin
   new(actual);
   new(needed);
   with g do
   begin
      { First: create the actual edge list }
      actual^.pred := actual;
      for i := 1 to NumVerts do
      begin
         LexOfI := vertices[i].LexNum;    { must translate }
         tempe := vertices[i].AdjList;
         while tempe^.next ~= nil do
         begin
            { Only need (i,j) where i < j }
            if vertices[tempe^.OtherEnd].LexNum > LexOfI then
               insert(actual, LexOfI,
                          vertices[tempe^.OtherEnd].LexNum);
            tempe := tempe^.next;
         end;
      end;
      actual^.pred^.suc := nil;

      { Create the list of necessary edges }
      needed^.pred := needed;
```

```
      for i := 1 to NumVerts do
           { Only look at labels with at least two entries }
         if vertices[i].cell^.flag^.LabelPtr ~= nil then
      begin
         tempc := vertices[i].cell^.flag;
         second := tempc^.LabelNum;
         tempc := tempc^.LabelPtr;
         while tempc^.LabelPtr ~= nil do
         begin
            insert(needed, second, tempc^.LabelNum);
            tempc := tempc^.LabelPtr;
         end;
      end;
      needed^.pred^.suc := nil;

      { Now snort the two lists }
      BucketSort(actual);
      BucketSort(needed);

      { See if needed is a subset of actual }
      act := actual^.suc;
      need := needed^.suc;
      CheckChordality := true;
      while need ~= nil do { Assume at least 1 actual edge }
      begin
         while (act^.v[1] < need^.v[1]) and (act^.suc ~= nil)
              do act := act^.suc;
         if act^.v[1] ~= need^.v[1] then
         begin
            CheckChordality := false;
            need := nil;
         end
         else
         begin
            while (act^.v[2] < need^.v[2]) and
               (act^.suc ~= nil) do act := act^.suc;
            if act^.v[2] = need^.v[2] then need := need^.suc
            else
            begin
               CheckChordality := false;
               need := nil;
            end;
         end; { of else }
     end; { of while }
   end; { with g }
end; { of function CheckChordality }
```

gr/./zsd/iso-test/chordal/main.p:

```
  { This is the main procedure for the chordality test. }

procedure main;
var     g:                            graph;
        outname:                      packed array[1..50] of char;
        outfile:                      text;

        procedure InputGraph(var g: graph); extern;
        procedure LexOrder(var g: graph; var out: text); extern;
        procedure OutputGraph(g: graph; var out: text); extern;
        function CheckChordality(g: graph): boolean; extern;

begin
   InputGraph(g);

   writeln('What is the output file name?');
   readln(outname);
   openf(outfile, outname, 'w');

   LexOrder(g, outfile);
   OutputGraph(g, outfile);
   if CheckChordality(g) then writeln('Graph is chordal!')
      else writeln('Graph is not chordal!');

   closef(outfile);
end;
```

Appendix 2

A Pascal Program for Isomorphism Testing

The following routines implement the tree isomorphism test. They are found under the catalog gr/./zsd/iso-test/iso.

gr/./zsd/iso-test/iso/types.p:

```
const   MaxVerts = 50;

type    NodeTypes = (P, Q, leaf);

        SonList = ^son;

        TreePtr = ^tree;

        son = record
        {begin}
           sibling:                       SonList;
           child:                         TreePtr;
        end;

        tree = record
        {begin}
           depth:                         integer;
           num:                           integer;
           INumber:                       integer;
           JNumber:                       integer;
           father:                        TreePtr;
           case NodeType: NodeTypes of
                P, Q:                     (sons:  SonList;
                                           symmetric:  boolean);
        end;


        list = ^ListElm;

        ListElm = record
        {begin}
           data:                          integer;
           next:                          list;
           case boolean of
              true: (position: integer);
              false: (tree: TreePtr);
        end;
```

```
        filename = packed array [1..40] of char;

        InfoArray = array [1..MaxVerts] of integer;

        InfoSet = array [1..MaxVerts] of list;

        BigInfoArray = array [1..2*MaxVerts] of integer;

        BigInfoSet = array [1..2*MaxVerts] of list;

        QueueHeader = ^qu;

        qu = record
        {begin}
           first:                        list;
           last:                         list;
        end;
```

gr/./zsd/iso-test/iso/stacks.p:

```
{ Stack functions follow...}

procedure push(var elm, lizst: list);
begin
   elm^.next := lizst;
   lizst := elm;
end;


function empty(lizst: list): boolean;
begin
   if lizst^.next = nil then empty := true else empty := false;
end;


function pop(var lizst: list): list;
begin
   pop := lizst;
   lizst := lizst^.next;
end;




{ Queue functions follow }

function NewQueueHeader: QueueHeader;
var     temp:                            QueueHeader;
begin
   new(temp);
   temp^.first := nil;
   temp^.last := nil;
   NewQueueHeader := temp;
```

```
end;


function QEmpty(Q: QueueHeader): boolean;
begin
   if Q^.first = nil then QEmpty := true
                      else QEmpty := false;
end;


function dequeue(var Q: QueueHeader): list;
begin
   dequeue := Q^.first;
   if Q^.first = Q^.last { we are emptying the queue }
      then Q^.first := nil
      else Q^.first := Q^.first^.next;
end;


procedure enqueue(elm: list;  var Q: QueueHeader);
begin
   if QEmpty(Q)
      then Q^.first := elm
      else Q^.last^.next := elm;
   Q^.last := elm;
end;


procedure concatenate(var Q1, Q2: QueueHeader);
begin
   if QEmpty(Q1)
      then Q1^.first := Q2^.first
      else Q1^.last^.next := Q2^.first;
   Q1^.last := Q2^.last;
   Q2^.first := nil;
   Q2^.last := nil;
end;
```

gr/./zsd/iso-test/iso/handlelist.p:

```
{ These functions are used to facilitate list handling. }

var     FreeList:                          list;

function empty(l: list): boolean;          extern;
function pop(var l: list): list;           extern;
procedure push(var l1, l2: list);          extern;

function GetList: list;
var     gnu:                               list;
begin
   if not empty(FreeList) then GetList := pop(FreeList)
   else
   begin
      new(gnu);
      GetList := gnu;
   end;
end;


function NewList: list;
var     temp:                              list;
begin
   temp := GetList;
   temp^.next := nil;
   NewList := temp;
end;


procedure GiveList(var lizst: list);
begin
   push(lizst, FreeList);
   lizst := nil;
end;


procedure InitializeFreeList;
begin
   new(FreeList);
   FreeList^.next := nil;
end;
```

gr/./zsd/iso-test/iso/utility.p:

```
function empty(l: list): boolean;        extern;
function pop(var l: list): list;         extern;
procedure push(var item, lst: list);     extern;

function NewQueueHeader: QueueHeader;    extern;
function QEmpty(Q: QueueHeader): boolean;   extern;
function dequeue(var Q: QueueHeader): list;   extern;
procedure enqueue(elm: list;   var Q: QueueHeader);   extern;
procedure concatenate(var Q1, Q2: QueueHeader);   extern;

function GetList: list;                  extern;
function NewList: list;                  extern;
procedure GiveList(var lst: list);       extern;
procedure InitializeFreeList;            extern;
```

gr/./zsd/iso-test/iso/get-g-info.p:

```
{ This procedure reads in the description of the graph
  which is output by the chordality test. }

procedure add(num: integer;  var l: list);
var     gnu:                            list;
begin
   gnu := GetList;
   gnu^.data := num;
   push(gnu, l);
end;

procedure BucketSort(var lst: list;
                     DataRange, PosRange: integer);   extern;

procedure GetGraphInfo(var NumVerts, NumCliques: integer;
                       ChordInfoFile: filename;
                       var degrees: InfoArray;
                       var DegSeqs: InfoSet);

var     kar:                            char;
        i, j:                           integer;
        ChordData:                      file of char;

begin
   openf(ChordData, ChordInfoFile, 'r');
   kar := ' ';
   while kar ~= 'V' do
         readln(ChordData, kar);   { Skip the headers }
   { Skip the * (first label maximal iff only 1 node) }
   read(ChordData, kar);
```

```
   readln(ChordData, NumVerts);

   while kar ~= 'N' do if eoln(ChordData) then readln(ChordData)
                              else readln(ChordData, kar);
   for i := 1 to NumVerts do
   begin
      for j := 1 to 2 do
      begin
         while kar ~= ':' do read(ChordData, kar);
         kar := ' ';
      end;
      readln(ChordData, degrees[i]);
   end;

   while kar ~= ' ' do if eoln(ChordData) then readln(ChordData)
                                    else readln(ChordData, kar);
   read(ChordData, i, j, NumCliques);

   reset(ChordData);
   while kar ~= 'V' do readln(ChordData, kar);
   for i := 1 to NumCliques do
   begin
      read(ChordData, kar);
      while kar = '*' do
      begin
         readln(ChordData);
         read(ChordData, kar);
      end;

      { We have a line with a clique on it }
      read(ChordData, j);
      kar := ',';
      DegSeqs[i] := NewList;
      while kar = ',' do
      begin
         read(ChordData, j, kar);
         add(degrees[j], DegSeqs[i]);
      end;
      readln(ChordData);
      BucketSort(DegSeqs[i], NumVerts, 0);
   end; { of for }

   closef(ChordData);
end; { of procedure GetGraphInfo }
```

gr/./zsd/iso-test/iso/inputree.p:

```
{ This procedure reads in the PQ-tree output by
  the interval graph test and builds a PQ-tree. }

function InputPQTree(PQfilename: filename): TreePtr;
var     pos:                            integer;
        depth:                          integer;
        NodeNum:                        integer;
        PQTree:                         packed array[1..200] of char;
        PQData:                         file of char;

{ This procedure adds a son to a tree }

procedure AddSon(var T, NewSon: TreePtr);
var     list:                               SonList;
begin
   new(list);
   list^.sibling := T^.sons;
   T^.sons := list;
   list^.child := NewSon;
end;

procedure SkipBlanks;
begin
   while PQTree[pos] = ' ' do pos := pos + 1;
end;

function GetInt: integer;
var     i:                                  integer;
begin
   i := 0;
   while (PQTree[pos] >= '0') and (PQTree[pos] <= '9') do
   begin
      i := i*10 + ord(PQTree[pos]) - 48;
      pos := pos + 1;
   end;
   GetInt := i;
end;

function BuildTree: TreePtr;
var     current:                            char;
        close:                              char;
        NewSon:                             TreePtr;
        T:                                  TreePtr;
begin
   depth := depth + 1;
   NodeNum := NodeNum + 1;
   SkipBlanks;

   { Create a null tree }
   new(T);
   T^.father := nil;
   T^.sons := nil;
```

```
   T^.depth := depth;
   T^.num := NodeNum;

   if PQTree[pos] = '[' then
   begin
      close := ']';
      T^.NodeType := Q;
   end
   else
   begin
      close := '>';
      T^.NodeType := P;
   end;
   pos := pos + 1;

   SkipBlanks;
   while PQTree[pos] ~= close do
   begin
      current := PQTree[pos];
      if (current = '[') or (current = '<')
      then NewSon := BuildTree
      else
      begin
         new(NewSon);
         NewSon^.NodeType := leaf;
         NewSon^.num := GetInt;
         NewSon^.depth:= depth + 1;
      end;
      NewSon^.father := T;
      AddSon(T, NewSon);
      SkipBlanks;
   end; { of while }
   pos := pos + 1;     { Skip the closing ] or > }
   depth := depth - 1;
   BuildTree := T;
end; { of procedure BuildTree }

begin { procedure InputPQTree }
   openf(PQData, PQfilename, 'r');
   while not eof(PQData) do readln(PQData, PQTree);
   closef(PQData);
   pos := 1;
   NodeNum := 0;
   depth := 0;
   InputPQTree := BuildTree;
end;
```

gr/./zsd/iso-test/iso/bucketsort.p:

```
{ This procedure bucketsorts a list of pairs;
  if PosRange is < 0 the list is sorted only on one field. }

procedure BucketSort(var llist: list;
                     DataRange, PosRange: integer);
var     bucket:                         BigInfoSet;
        current:                        list;
        i:                              integer;

procedure move(var item, into: list);
var     temp:                           list;
begin
   temp := item^.next;
   push(item, into);
   item := temp;
end;

begin
   for i := 1 to DataRange do bucket[i] := NewList;
   current := llist;
   while current^.next ~= nil do
                    move(current, bucket[current^.data]);

   llist := current;
   for i := 1 to DataRange do
      while bucket[i]^.next ~= nil do move(bucket[i], llist);

   for i := 1 to DataRange do GiveList(bucket[i]);

   if PosRange > 0 then
   begin
      for i := 1 to PosRange do bucket[i] := NewList;
      current := llist;
      while current^.next ~= nil do
                    move(current, bucket[current^.position]);

      llist := current;
      for i := 1 to PosRange do
         while bucket[i]^.next ~= nil do move(bucket[i], llist);

      for i := 1 to PosRange do GiveList(bucket[i]);
   end;
end; { of procedure BucketSnort }
```

gr/./zsd/iso-test/iso/vlssort.p:

```
{ Variable Length String Sort - we use a modified lexicographic
  sort algorithm to "sort" an array of lists;  each list is a
  sequence of integer.
  The ranks are returned in "rank".
  The lists are passed in "strings".
  "tuples" tells how many lists there are to be sorted.
  "largest" gives the largest integer appearing in any tuple.
}

procedure BucketSort(var lst: list;
                     DataRange, PosRange: integer);   extern;

procedure VLSSort(strings: BigInfoSet;  var rank: BigInfoArray;
                  tuples, largest: integer);

var     NonEmpty:             array[1..2*MaxVerts] of QueueHeader;
        Q:                    array[1..2*MaxVerts] of QueueHeader;
        queue:                QueueHeader;
        NE:                   list;
        elm:                  list;
        temp:                 list;
        tmp2:                 list;
        current:              list;
        pointers:             BigInfoSet;
        MaxLength:            integer;
        i, l:                 integer;
        count:                integer;
        previous:             integer;
begin
   { First make a list of all pairs
     of (list number) vs (list data) }
   NE := NewList;
   MaxLength := 0;
   for i := 1 to tuples do
   begin
      count := 0;
      elm := strings[i];
      while elm^.next ~= nil do
      begin
         count := count + 1;
         temp := GetList;
         temp^.data := elm^.data;
         temp^.position := count;
         push(temp, NE);
         elm := elm^.next;
      end;
      if count > MaxLength then MaxLength := count;
   end;

   { Now sort this list of pairs }
   BucketSort(NE, largest, MaxLength);
```

```
{ Now create the NonEmpty lists }
for i := 1 to MaxLength + 1 do NonEmpty[i] := NewQueueHeader;
current := NE;
while current^.next ~= nil do
begin
   temp := current;
   current := current^.next;
   { If the list element following temp is the same as temp,
     ignore temp }
   if (temp^.position = current^.position) and
      (temp^.data = current^.data) and (current^.next ~= nil)
      then GiveList(temp)
      else enqueue(temp, NonEmpty[temp^.position]);
end;

{ Now we are (finally!) ready to start sorting }
previous := 0;
queue := NewQueueHeader;
for i := 1 to tuples do
begin
   pointers[i] := strings[i];
   new(temp);
   temp^.data := i;
   enqueue(temp, queue);
end;
for i := 1 to largest do Q[i] := NewQueueHeader;
for l := 1 to MaxLength + 1 do
begin
   while not QEmpty(queue) do
   begin
      current := dequeue(queue);
      if pointers[current^.data]^.next = nil
      then begin
         { This tuple is "sorted"...find it's rank }
         if previous = 0
         then begin
            previous := current^.data;
            rank[current^.data] := 1;
         end
         else begin
            { Compare this string for equality
              with the previous string }
            temp := strings[previous];
            tmp2 := strings[current^.data];
            while (temp^.next ~= nil) and (tmp2^.next ~= nil)
               and (temp^.data = tmp2^.data) do
            begin
               temp := temp^.next;
               tmp2 := tmp2^.next;
            end;
            if (temp^.next = nil) and (tmp2^.next = nil)
               then rank[current^.data] := rank[previous]
               else begin
                  rank[current^.data] := rank[previous] + 1;
```

```
                    previous := current^.data;
                 end;
              end; { of long else clause }
          end { of big then clause }
          else begin
             enqueue(current, Q[pointers[current^.data]^.data]):
             pointers[current^.data] :=
                                pointers[current^.data]^.next;
          end;
       end; { of while }

       { Concatenate all of the appropriate queues together }
       while not QEmpty(NonEmpty[l]) do
       begin
          current := dequeue(NonEmpty[l]);
          concatenate(queue, Q[current^.data]);
          GiveList(current);
       end;
    end; { of big for l := 1 to MaxLength }
end; { of procedure VLSSort }
```

gr/./zsd/iso-test/iso/i-number.p:

```
procedure VLSSort(strings:BigInfoSet;  var rank: BigInfoArray;
                  tuples, largest: integer);   extern;

procedure BucketSort(var lst: list;
                     DataRange, PosRange: integer); extern;

function number(DS1, DS2:  InfoSet;
                T1, T2:  TreePtr;
                NumVerts, NumVert2:  integer;
                NumCliques, NumClique2:  integer):  boolean;

var     DegSeqs:                             BigInfoSet;
        i:                                   integer;
        ranks:                               BigInfoArray;
        levels:                              InfoSet;
        temp:                                list;

procedure GetLevels(var T: TreePtr;  start: integer);
var     son:                                 SonList;
        temp:                                list;
begin
   { This procedure creates an entry in the list 'level[i]' for
     each vertex of T whose depth is i.   Also, given the rank
     information a working label is given to each leaf.  }

   temp := GetList;
```

```
   temp^.tree := T;
   push(temp, levels[T^.depth]);

   if T^.NodeType ~= leaf
      then
      begin
         son := T^.sons;
         while son ~= nil do
         begin
            GetLevels(son^.child, start);
            son := son^.sibling;
         end;
      end
      else if start >= 0 then T^.INumber := ranks[T^.num + start];
end;  { of procedure GetLevels }

procedure INumber;
var     l:                                 integer;
        largest:                           integer;
        strings:                           integer;
        i:                                 integer;
        MaxDepth:                          integer;
        T:                                 TreePtr;
        labels:                            BigInfoSet;
        son:                               SonList;
        node:                              list;
        symbol:                            list;
        temp:                              list;
        stak:                              list;
begin
   { Create the lists of nodes at each level }
   GetLevels(T1, 0);
   GetLevels(T2, NumCliques);
   MaxDepth := NumCliques;
   while empty(levels[MaxDepth]) do MaxDepth := MaxDepth - 1;
   stak := NewList;

   { Do the i-numbering from bottom of tree to top }
   for l := MaxDepth downto 1 do
   begin
      { Create labels at this level }
      strings := 0;
      largest := 2;
      node := levels[l];
      while node^.next ~= nil do
      begin
         T := node^.tree;
         strings := strings + 1;
         labels[strings] := NewList;
         if T^.NodeType = leaf then
         begin
            temp := GetList;
            temp^.data := T^.INumber;
            push(temp, labels[strings]);
```

```
    if T^.INumber > largest then largest := T^.INumber;
end
else
begin
    son := T^.sons;
    while son ~= nil do
    begin
        temp := GetList;
        temp^.data := son^.child^.INumber;
        if temp^.data > largest then largest := temp^.data;
        push(temp, labels[strings]);
        son := son^.sibling;
    end;

    { P and Q nodes handled differently wrt label }
    if T^.NodeType = P
        then BucketSort(labels[strings], 2*NumVerts, 0)
        else
        begin
            { Choose the smaller of the label
              and it's reversal. }
            symbol := labels[strings];
            while symbol^.next ~= nil do { Create reversal }
            begin
                temp := GetList;
                temp^.data := symbol^.data;
                push(temp, stak);
                symbol := symbol^.next;
            end;

            { Compare them }
            T^.symmetric := true;
            symbol := labels[strings];
            temp := stak;
            while (temp^.next ~= nil) and T^.symmetric do
            begin
                if temp^.data ~= symbol^.data then
                begin
                    T^.symmetric := false;
                    if temp^.data < symbol^.data then
                    begin
                        temp := labels[strings];
                        labels[strings] := stak;
                        stak := temp;
                    end;
                end;
                symbol := symbol^.next;
                temp := temp^.next;
            end;
            while not empty(stak) do
            begin
                temp := pop(stak);
                GiveList(temp);
            end;
```

```
                  end; { of else }

               { Add a P/Q indicator }
               temp := GetList;
               if T^.NodeType = P
                  then temp^.data := 1
                  else if T^.NodeType = Q then temp^.data := 2;
               push(temp, labels[strings]);
            end; { of else clause }
            node := node^.next;
         end; { of while }

         { Sort the labels and assign the i-numbers }
         VLSSort(labels, ranks, strings, largest);
         strings := 0;
         while not empty(levels[l]) do
         begin
            strings := strings + 1;
            node := pop(levels[l]);
            node^.tree^.INumber := ranks[strings];
            GiveList(node);
         end;

         { Give back the storage used by the labels }
         for i := 1 to strings do
         begin
            while not empty(labels[i]) do
            begin
               temp := pop(labels[i]);
               GiveList(temp);
            end;
            GiveList(labels[i]);
         end;
      end; { of for l := .. }
   end; { of procedure INumber }

   begin { function number }
      if (NumVerts ~= NumVert2) or (NumCliques ~= NumClique2)
         then number := false
         else
         begin
            { Concatenate the clique degree sequences and sort them }
            for i := 1 to NumCliques do
            begin
               DegSeqs[i] := DS1[i];
               DegSeqs[NumCliques + i] := DS2[i];
            end;
            VLSSort(DegSeqs, ranks, 2*NumCliques, NumVerts);

            { Initialize the lists of nodes at each level }
            for i := 1 to NumCliques      { really only need about }
               do levels[i] := NewList;   { log(NumCliques) }

            { And do the i-numbering }
```

```
        INumber;

        { Compare the i-numbers of the roots of T1 and T2 }
        number := T1^.INumber = T2^.INumber;
     end; { of else }
end; { of function number }
```

gr/./zsd/iso-test/iso/main.p:

```
function InputPQTree(f: filename): TreePtr;     extern;

procedure GetGraphInfo(var i,j: integer; f: filename;
                       var d: InfoArray;
                       var ds: InfoSet);     extern;

function number(DS1, DS2: InfoSet;
                T1, T2: TreePtr;
                NumVerts, NumVert2: integer;
                NumCliques, NumClique2: integer): boolean;
                                                   extern;

procedure main;
var     PQTree1:                        TreePtr;
        PQTree2:                        TreePtr;
        ChordOutputData1:               filename;
        ChordOutputData2:               filename;
        PQOutputData1:                  filename;
        PQOutputData2:                  filename;
        degrees1:                       InfoArray;
        degrees2:                       InfoArray;
        DegSeqs1:                       InfoSet;
        DegSeqs2:                       InfoSet;
        NumVerts1:                      integer;
        NumVerts2:                      integer;
        NumCliques1:                    integer;
        NumCliques2:                    integer;
        isomorphic:                     boolean;

begin
   InitializeFreeList;

   writeln('What is  1 chordality test output file name?');
   readln(ChordOutputData1);
   writeln('What is  1 PQ output file name?');
   readln(PQOutputData1);

   GetGraphInfo(NumVerts1, NumCliques1, ChordOutputData1,
                degrees1, DegSeqs1);
```

```
    PQTree1 := InputPQTree(PQOutputData1);

    writeln('What is  2 chordality test output file name?');
    readln(ChordOutputData2);
    writeln('What is  2 PQ output file name?');
    readln(PQOutputData2);

    GetGraphInfo(NumVerts2, NumCliques2, ChordOutputData2,
                 degrees2, DegSeqs2);

    PQTree2 := InputPQTree(PQOutputData2);

    isomorphic := number(DegSeqs1, DegSeqs2, PQTree1, PQTree2,
                         NumVerts1, NumVerts2, NumCliques1,
                         NumCliques2);
    if isomorphic then writeln('Graphs are isomorphic!')
         else writeln('Graphs are not isomorphic');
end;
```

Appendix 3

How to Access and Run the Isomorphism Test

The entire set of programs and all (available) related information for the interval graph isomorphism test is found under the catalog 'gr/./zsd/iso-test' on the University of Waterloo Math Faculty Computing Facility Honeywell 66/60. If this catalog is partially or wholly non-existent, it can be restored (by someone with appropriate privileges) by typing 'arch r gr/./zsd/iso-test'. This catalog should have read and execute permissions on it; i.e. any user should be able to use the programs.

There are two methods one might use to run the isomorphism test. The first is to type 'gr/./zsd/iso-test/exec.ec' which will run the programs in the correct order, ask the appropriate questions, and insure that the files are correctly specified. This method suffers from the 'asking redundant questions' illness; this can lead to a mild case of tedium.

The second method is to type all of the appropriate command lines directly. The method suffers from the dreaded 'user must have a basic idea of what he is doing' disease. Both of the methods are shown in the examples in Appendix 4.

Using either method, the user must know how to specify a filename. On the Honeywell, a <u>temporary</u> file (which disappears when the user signs off) is specified by one to

eight letters, digits, periods and some other characters. For example, 'graph1', 'new.data' and 'g' are all valid temporary file names. Any valid temporary file name (as well as others) preceded by a '/' specifies a permanent file; these files do not disappear when the user signs off.

The only other critical part of running the isomorphism test is the input format for the graphs. The data can be interactively typed in while running the chordality test or it can be stored in a file. The latter is the preferred choice; in the former case the data is lost after the chordality test finishes, and if an error is made the user will have to start over from the beginning. The format for typing in data can be found in Appendix 1 in the 'input' subroutine and also in the examples in Appendix 4.

Appendix 4

Examples

In the following examples, both methods of running the isomorphism test (see Appendix 3) are displayed. The column of c's (for computer), b's (for both) and U's (for user) is not typed out at any time; it was added to distinguish lines types solely by the user from those typed solely by the user from those typed by the computer and user. All input graphs used here are stored under the catalog 'gr/./zsd/iso-test'; see Appendix 3 for the details of accessing these. The graphs all have file names like 'examplel'.

The first example demonstrates the use of both gr/./zsd/iso-test/exec.ec and typing in data interactively. The program tells the user that the graph is not chordal; thus the graph is not an interval graph and is not amenable to this isomorphism test.

```
U ec gr/./zsd/iso-test/exec.ec
c First graph:
c If data in a file, give filename; else hit return
b How many vertices (max = 50) ?? 4
c List adjacent vertices for each vertex; end with a 0
b 1: 2 4 0
b 2: 1 3 0
b 3: 2 4 0
b 4: 1 3 0
c
c  *** input complete ***
c
c What is the output file name?
U graph
c Graph is not chordal!
c Do you want to stop? (y or n)
U y
```

The second example shows input being taken from a file. The first graph happens to be a chordal graph, but not an interval graph. This is signified in the print out of the tree by the phrase 'before rejection'. The data for this graph is as follows:

```
6
3 4 0
3 5 0
1 2 4 5 0
1 3 5 6 0
2 3 4 6 0
4 5 0
```

```
U ec gr/./zsd/iso-test/exec.ec
c First graph:
c If data in a file, give filename; else hit return
U gr/./zsd/iso-test/chordal/chordal.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph1
c Graph is chordal!
c Do you want to stop? (y or n)
U n
c Second graph:
c If data in a file, give filename; else hit return
U gr/./zsd/iso-test/chordal/rtl.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph2
c Graph is chordal!
c Do you want to stop? (y or n)
U n
c What was the first graph output file again?
U graph1
c And the second again?
U graph2
c The data from the chordality tests have been modified
c and are now in 'massage1' and 'massage2' respectively.
c
c Type in massage1
U massage1
c Type in massage2
U massage2
c The PQ tree for the first graph has been strored in 'tree1'.
```

```
c Here it is:
c And the data file name is:
c
c  for the set of sets
c
c   1 (   4 )
c   2 (   3 )
c   3 (   2   3   4 )
c   4 (   1   2   4 )
c   5 (   1   2   3 )
c   6 (   1 )
c  the last structure before rejection
c  [   3 <   4    2 >   1 ]
c Do you want to stop? (y or n)
U y
```

The third example shows two non-isomorphic interval graphs. The two data files follow.

```
gr/./zsd/iso-test/chordal/intrval1.dat:
7
2 0
1 3 4 5 0
2 5 0
2 5 0
2 3 4 6 0
5 7 0
6 0

gr/./zsd/iso-test/chordal/intrval3.dat:
7
2 0
1 3 4 5 0
2 5 0
2 5 0
2 3 4 6 7 0
5 0
5 0

U ec gr/./zsd/iso-test/exec.ec
c First graph:
c If data in a file, give filename; else hit return
U gr/./zsd/iso-test/chordal/intrval1.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph1
c Graph is chordal!
c Do you want to stop? (y or n)
U n
c Second graph:
```

```
c If data in a file, give filename; else hit return
U gr/./zsd/iso-test/chordal/intrval3.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph2
c Graph is chordal!
c Do you want to stop? (y or n)
U n
c What was the first graph output file again?
U graph1
c And the second again?
U graph2
c The data from the chordality tests have been modified
c and are now in 'massage1' and 'massage2' respectively.
c
c Type in massage1
U massage1
c Type in massage2
U massage2
c The PQ tree for the first graph has been strored in 'tree1'.
c Here it is:
c And the data file name is:
c
c  for the set of sets
c
c   1 (   5 )
c   2 (   3   4   5 )
c   3 (   4 )
c   4 (   3 )
c   5 (   2   3   4 )
c   6 (   1   2 )
c   7 (   1 )
c temp2 is nil
c oops  1
c oops  2
c oops  3
c  [   5  <   4   3 >   2   1 ]
c Do you want to stop? (y or n)
U n
c The PQ tree for the second graph has been stored in 'tree2'.
c Here it is:
c And the data file name is:
c
c  for the set of sets
c
c   1 (   5 )
c   2 (   3   4   5 )
c   3 (   4 )
c   4 (   3 )
c   5 (   1   2   3   4 )
c   6 (   2 )
c   7 (   1 )
```

```
c  [    5  <   4    3 > <   1    2 >]
c Do you want to stop? (y or n)
U n
c
c Now just answer the following questions with the appropriate
c filenames.
c What is  1 chordality test output file name?
U graph1
c What is  1 PQ output file name?
U tree1
c What is  2 chordality test output file name?
U graph2
c What is  2 PQ output file name?
U tree2
c Graphs are not isomorphic
```

The final example shows two isomorphic interval graphs as well as the method of using the isomorphism test without the 'exec.ec' file. In reality, the 'tree1' and 'tree2' files should be printed and examined, to insure that the graphs are both interval graphs. If this check is not made and the tree isomorphism test is run, for non-interval graphs the result will (in general) be meaningless. The file gr/./zsd/iso-test/chordal/intrval1.dat is shown above. The other file follows.

```
gr/./zsd/iso-test/chordal/intrval2.dat:
7
5 7 0
5 0
5 7 0
6 7 0
1 2 3 7 0
4 0
1 3 4 5 0
```

```
U gr/./zsd/iso-test/chordal/chord.lm
c If data in a file, give filename; else hit return
U jsdiamond/iso-test/chordal/intrval1.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph1
c Graph is chordal!

U gr/./zsd/iso-test/chordal/chord.lm
c If data in a file, give filename; else hit return
U jsdiamond/iso-test/chordal/intrval2.dat
c
c  *** input complete ***
c
c What is the output file name?
U graph2
c Graph is chordal!

U gr/./zsd/iso-test/interval/massg.lm <graph1 >massage1
U gr/./zsd/iso-test/interval/massg.lm <graph2 >massage2

U gr/./zsd/iso-test/interval/pqtre.lm >tree1
U massage1
U gr/./zsd/iso-test/interval/pqtre.lm >tree2
U massage2

U gr/./zsd/iso-test/iso/itest.lm
c What is  1 chordality test output file name?
U graph1
c What is  1 PQ output file name?
U tree1
c What is  2 chordality test output file name?
U graph2
c What is  2 PQ output file name?
U tree2
c Graphs are isomorphic!
```