SETL Newsletter #125

Schaefer's Node Splitting Algorithm

K. Kennedy, Rice University

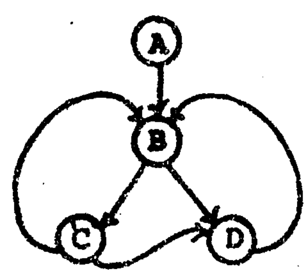
February 6, 1974

In his book [1], Schaefer introduces a node splitting algorithm which is slightly different from that of Cocke. This newsletter produces a SETL version of Schaefer's algorithm with a somewhat modified narrative discourse. The algorithm has two parts: first, given an irreducible graph, prime cycles are located and a set of nodes which can break these cycles is chosen; second, the set of nodes are used as interval heads for the reduction and the split graph (which will allow intervalization) is constructed along with its successor relation.

Finding Prime Cycles

The purpose of this section is to select a set of nodes which, when removed from the graph will leave it acyclic. Certainly a set of nodes which break all *prime cycles* -- cycles which contain no subcycles -- will do.

In the graph below



the prime cycles are [B,C] and [B,D] while [B,C,D] is what we call an *elementary* or *simple* cycle, that is, a simple path from B to an immediate predecessor of B. The cycle [B,C,D] is not prime however since it contains both [B,C] and [B,D]. Our approach to finding prime cycles will be to enumerate all elementary cycles and to eliminate those which are not prime.

One way to enumerate elementary cycles would be to use a recursive backtracking algorithm which looks for simple paths to a selected node by working back from that node through the predecessor relation. However, for large graphs such an algorithm could be exorbitant in cost, so we must find ways of limiting our search.

The approach we shall use is the "method of elementary developments" due to B. Roy [2]. This method uses an auxiliary table, which can be constructed rapidly, to cut down search time. Suppose that in a given graph we are searching for elementary cycles which include the node *b* but do not include the entry node *a* of the graph and suppose that

nodes = {c,d,e,f,g,h}

are the remaining nodes in the graph. The auxiliary table will be a square matrix with dimension equal to the number of remaining nodes -- it will have one entry for each remaining node and one entry for each possible path length (in number of arcs). Thus for a given node *x* and a given path length *i*,

```
table(i,x) = {n ∈ graph - {a} |
   ∃a simple path with i arcs from b to x
   whose final arc is from n to x}
```

Several facts about this table should be noted.

1) *table(1,x)* = {*b*} if *b* is a predecessor of *x* and <u>nℓ</u> otherwise, because the only path of length 1 from b to x is the arc from b to x.
2) *table(i,x)* must be contained in the set of predecessors of *x*; otherwise there can be no arc from a node *n* ∈ *table(i,x)* to *x*;
3) If *n* ∈ *table(i,x)* then *table(i-1,n)* must not be empty because if there is a simple path from *b* to *x* of length *i* whose last arc is from *n* to *x*, there must be a simple path of length *i*-1 from *b* to *n*.
4) Any simple path from *b* to *x* must not contain *x* (except as the last node), otherwise the path is not simple.

These facts will be used to construct the auxiliary table.

One of our basic tools will be an algorithm which backtracks through the table to produce all simple paths of length *i* from some node *b* to another node *x* which does not contain any of the nodes in the set *notcontaining*. The method is trivial: there is a simple path of length *i* from *b* to *x* if there exists a node *n* such that there is a simple path of length *i*-1 to *n* (not containing *x*) and an arc from *n* to *x*. The following routine embodies this method.

```
definef simplepaths(b,x,i,notcontaining)
/* table is global to this routine */
/* first find out if there are any such paths */
   spaths = nℓ;
   if i eq 1 and table(1,x) ne nℓ
      then return {<b,x>};  end if;
   if (table(i,x) - notcontaining) eq nℓ
      then return nℓ;  end if;
   /* backtrack recursively */
   (∀y ∈ (table(i,x) - notcontaining))
      not = notcontaining with y;
      paths = simplepaths(b,y,i-1,not);
      (∀z ∈ paths) spaths = spaths + {z + <x>};;
   end ∀y;
   return spaths;
end simplepaths;
```

This routine can be used to check condition 4 above when building the auxiliary table.

The auxiliary table can be built by setting the first row according to condition 1, then using conditions 2, 3 and 4 to construct other rows. The following SETL code fragment expresses this notion.

```
table(i,x) = {n ∈ nodes |
   /* condition 2 */  n ∈ pred(x) and
   /* condition 3 */  table(i-1,n) ne nl  and
   /* condition 4 */
     simplepaths(b,n,i-1,notcontaining+{x,n}) ne nl}
```

Elementary cycles can then be enumerated by looking at all predecessors *x* of *b* and adding all simple paths from *b* to *x* to the list.

One problem remains, however. Once we have found the list of elementary cycles which pass through *b*, we must find elementary cycles which do not pass through *b*. We note a simple fact: there is a simple path from node *c* to *x* if there is a simple path from *b* to *x* whose first arc is from *b* to *c*, i.e., if we are to use the information on simple paths from *b* to determine simple paths from *c*, *c* must be an immediate successor of *b* and *c* must have successors which are in the remaining nodes of the graph. Thus we can derive information about simple paths from *c* by modifying the table for *b*. The modification is performed as follows.

1. Pick a node *c* which meets the requirements stated above.
2. Remove the column corresponding to *c* from the table.
3. Remove the first row of the table.
4. Renumber the remaining rows from 1.
5. Eliminate all elements but *c* from the new first row.
6. Eliminate all instances of *c* from other rows.
7. Apply condition 3 to each of the rows except the first, eliminating further elements.

The following function performs these modifications. Its result is the new node *c* but it also modifies the arguments *table*, *notcontaining*, and *nodes*.

```
definef modify (table, b, notcontaining, nodes);
   /* pred and succ are global */
   /* first find a new pivot element */
   c = ∋{n ∈ nodes | c ∈ succ(b) and succ(c) * nodes ne nℓ};
   nodes = nodes - c;
   notcontaining = nodes with c;
   /* now redefine table */
   (∀x ∈ nodes)  /* first row */
      newtable(1,x) = if x ∈ succ(c)
                           then {c} else nℓ;  end ∀x;
   /* remaining rows */
   (2 ≤ ∀i ≤ #nodes, ∀x ∈ nodes)
      newtable(i,x) = {y ∈ table (i+1,x) |
         /* condition 3 */
                newtable(i-1,y) ne nℓ};
   end ∀i;
   table = newtable;
   return c;
end modify;
```

Finally, once we have all the elementary cycles we can enumerate the prime cycles by eliminating cycles which contain other cycles. The algorithm below chooses the shortest elementary cycles as prime cycles first; then it adds cycles of increasing length (which do not contain other cycles).

```
definef findprimes(elemcycles);
   /* find minimum and maximum length */
   minlength = [min: c ∈ elemcycles] #c;
   maxlength = [max: c ∈ elemcycles] #c;
   /* cycles of minimum length must be prime */
   primes = {c ∈ elemcycles|#c = minlength};
   minlength = minlength + 1;
   /* add more cycles while increasing length */
   (while minlength le maxlength doing
               minlength = minlength + 1;)
      (∀c ∈ elemcycles|#c = minlength)
         if n(∃cycle ∈ primes |
            (1 ≤ ∀i ≤ #cycle, 1 ≤ ∃j ≤ #c|cycle(i)=c(j)))
            then primes = primes with c;;
      end ∀c;
   end while;
   return primes;
end findprimes;
```

We are now ready to present the complete algorithm for finding prime cycles. The input to this algorithm is a graph, i.e. a set of nodes *graph*, an entry node *entry*, the successor relation *succ*, and the predecessor relation *pred*.

```
definef findprimecycles(graph,entry,succ,pred);
   /* first initialize the set of nodes which cannot be in a cycle*/
   notcontaining =  entry ;
   nodes = graph less entry;
   /* select the node b --- to enumerate cycles through b */
   b =  ∋{n ∈ nodes | n ∈ succ(entry) and
                              succ(n) * (nodes less n) ne nℓ};
   nodes = nodes less b;
   notcontaining = notcontaining with b;
   /* set up the auxiliary table */
   table = nℓ;
   /* first row */
   (∀x ∈ nodes)
      table(1,x) = if b ∈ pred(x) then {b} else nℓ;
   end ∀x;
   /* now construct the remaining rows using conditions 2,3 and 4*/
   (2 ≤ ∀i ≤ #nodes, ∀x ∈ nodes)
      table(i,x) = {n ∈ nodes |
         /* condition 2 */  n ∈ pred(x) and
         /* condition 3 */  table(i-1,n) ne nℓ and
         /* condition 4 */
            simplepaths(b,n,i-1,notcontaining with x) ne nℓ};
   /* next the loop to enumerate elementary cycles */
   elemcycles = nℓ;
   (while #nodes ge 2 doing /* modify table */
              b = modify(table,b,notcontaining, nodes);
              /* b,table,notcontaining,nodes are changed*/)
      (2 ≤ ∀length ≤ #nodes, ∀x ∈ (pred(b) * nodes))
         elemcycles = simplepaths (b,x,length,notcontaining);
      end ∀length;
   end while;
   /* reduce to prime cycles and return */
   return (findprimes(elemcycles));
end findprimecycles;
```

The reader should note that this routine can be improved in efficiency if non-prime elementary cycles are eliminated as they are added, since these cycles are added in order of increasing length.

Factor Sets

We define a set of nodes F to be a *factor set* if the removal of the nodes in F from the graph will make the graph cycle-free. A factor set is said to be *minimal* if it contains no factor set as a proper subset. In the algorithm to follow, a minimal factor set is used to split the graph, so we now present an algorithm which produces such a set.

Suppose *primecycles* is the set of prime cycles found by our previous algorithm. We can construct a minimal factor set by picking an arbitrary element from one of the prime cycles and removing from *primecycles* all cycles which contain that element, repeating until *primecycles* is exhausted.

```
definef minfact(primecycles);
   minset = nl;
   p = primecycles;
   /* loop until p is exhausted */
   (while p ne nl)
      x from p;  elt = x(1);
      minset = minset with elt;
      /* remove cycles with elt */
      p = p - {c ∈ primecycles |
               (1 ≤ ∃i ≤ #c|c(i) eq elt};
   end while p;
   return minset;
end minfact;
```

The set returned by this routine is clearly a factor set since any prime cycle contains at least one element in the set (breaking prime cycles is sufficient to break the graph because every cycle contains a prime cycle). The question is: is this set minimal? Suppose that it is not; then there is a proper subset which is also a factor set. In particular, if

$$\text{minset} = \{e_1, e_2, \ldots, e_n\} ,$$

where the elements are numbered in the order they are added by the algorithm, then some element, say e_j , is not in the subset. Then all cycles which contain e_j must also contain some e_i , $i \neq j$. Let c be some prime cycle which contains two of the elements of minset. Assume $i < j$, then all prime cycles containing e_i have been removed before e_j is selected so this is impossible. The same argument works for $i > j$ and we have the desired contradiction. The factor set returned by the above algorithm must be minimal.

Splitting the Graph

The basic idea of Schaeffer's method is to pick a minimal factor set for the irreducible graph and use these nodes as interval heads for intervals on the next level. In other words, we will force the graph to reduce by splitting nodes which might be in an interval tending from one of these nodes. In the split graph there will be one copy of each of the interval heads but each interval head will have its own copy of any node that can be reached from that head by a path which does not include another interval head. The interval head along with its copies of nodes in the split graph will form an interval for the reduction step.

Nodes in the split graph are denoted (in the notation of Schwartz [3]) by ordered pairs. If h is an interval head, the pair $\langle h,h \rangle$ will represent h in the split graph; if b is a node (not a head) which can be reached from h by a path which does not include another head then h's copy of b is denoted by $\langle b,h \rangle$. There may be several copies of b in the split graph belonging to several different heads.

The successor function for the split graph is constructed in the natural way. Suppose $<b,h_1>$ and $<b,h_2>$ are two copies of b which belong to the two heads h_1 and h_2 respectively. If, in the original irreducible graph s (not a head) is a successor of b then there must necessarily be two copies $<s,h_1>$ and $<s,h_2>$ of s in the split graph and $<s,h_1>$ is a successor of $<b,h_1>$ while $<s,h_2>$ is a successor of $<b,h_2>$. If h_3 (an interval head) is a successor of b in the original graph, $<h_3,h_3>$ is a successor of <u>both</u> $<b,h_1>$ and $<b,h_2>$ in the split graph. This method of constructing the successor function assures us that any copy (in the split graph) of a node in the original graph will be able to branch to <u>at least one copy</u> in the split graph of each of its successors in the original graph -- a requirement if the split graph is to be equivalent to the original graph.

We now present the general method for constructing the split graph.

1. Initially let the set of interval heads be the minimal factor set augmented by the graph entry node.
2. For every h in the set of heads, construct the interval for h as follows.
   a. The head node $<h,h>$ is added to the nodes of the split graph.
   b. The set of nodes I in the interval (the nodes in the original graph with copies in the interval) is initially {h}.
   c. For each node s which is a successor in the original graph of some b in I, if s is not an interval head construct $<s,h>$ and add it to the nodes of the graph. Add the node s to I.

3. Construct the successor function for the split graph as follows: for each pair $\langle x,y \rangle$ in the split graph consider each successor, say s, of x in the original graph.
   a. If s is an interval head, make $\langle s,s \rangle$ a successor of $\langle x,y \rangle$ in the split graph.
   b. If s is not an interval head, make $\langle s,y \rangle$ a successor of $\langle x,y \rangle$ in the split graph.

Schaefer [1] has shown that the resulting split graph will reduce to a graph with fewer nodes than in the original irreducible graph. This process can be applied repeatedly until the graph finally reduces to a single node.

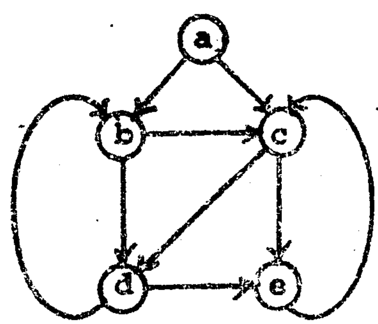
We now present the SETL version of Schaefer's algorithm. It accepts as input a graph of the form $\langle nodes, pred, succ, entry \rangle$ and it returns the split graph in the same form.

```
definef splitnodes(graph);
   /* first break up the graph */
   <nodes,succ,pred,entry> = graph;
   primecycles = findprimecycles(nodes,succ,pred,entry);
   minset = minfact(primecycles);
   /* now construct the set of heads by adding the entry node
                                                   to minset */
   heads = minset with entry;
   /* apply the method described in this section to copy nodes
      and construct the succ and pred functions */
   newnodes = nℓ;
   (∀h ∈ heads)
      int = {<h,h>};
      newnodes = newnodes with <h,h>;
      (while int ne nℓ)
         n from int;
         /* look at successors to construct successor  fn */
```

```
          (∀x ∈ succ(hd n))
            if x ∈ heads then
              succ{n} = succ{n} with <x,x>;
              pred{<x,x>} = pred{<x,x>} with n;
            else
              succ{n} = succ{n} with <x,h>;
              pred{<x,h>} = pred{<x,h>} with n;
            /* avoid treating x twice */
            if <x,h> n ∈ newnodes then
              newnodes = newnodes with <x,h>;
              int = int with <x,h>;
            end if x;
          end ∀x;
        end while int;
      end ∀h;
      return <newnodes, succ, pred, <entry,entry>>;
end splitnodes;
```

An Example

Consider the following irreducible graph



The prime cycles are <b,d> and <c,e>. We choose {b,c} as the minimum factor set and {a,b,c} becomes the set of heads. First we choose *a* from the set of heads and get

nodes: <a,a>
succ{<a,a>} = {<b,b>,<c,c>}

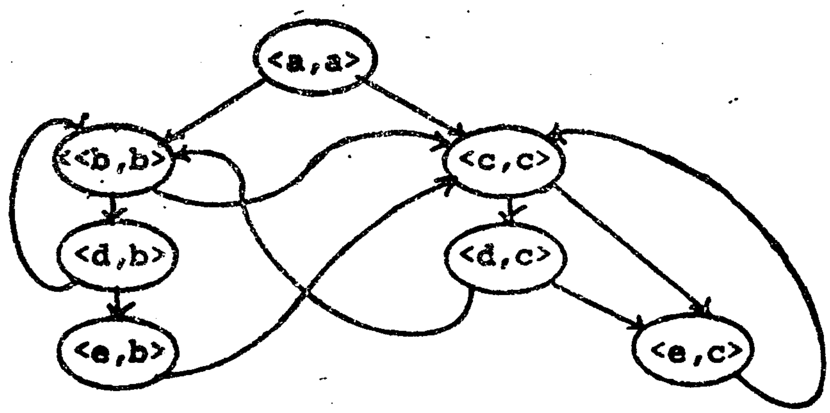
Next *b*.

nodes: <b,b>, <d,b>, <e,b>
succ{<b,b>} = {<c,c>,<d,b>}
succ{<d,b>} = {<b,b>,<e,b>}
succ{<e,b>} = {<c,c>}

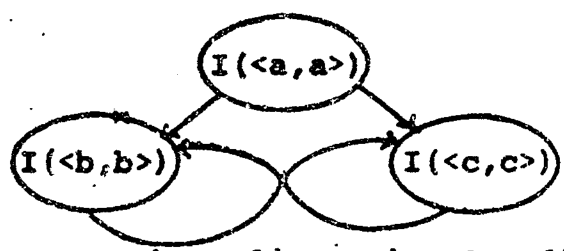
Next *c*.

nodes: <c,c>, <d,c>, <e,c>
succ{<c,c>} = {<d,c>,<e,c>}
succ{<d,c>} = {<b,b>,<e,c>}
succ{<e,c>} = {<c,c>}

The new graph becomes



which reduces to



which must, of course, be split again. One disadvantage of this method is the proliferation of split nodes which may make it less practical than Cocke's algorithm.

Acknowledgement. This work was supported by the National Science Foundation, Office of Computing Activities, Contract NSF-GJ-40585.

References

[1] Schaefer, M. A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] Roy, B., Cheminement et Connexité dans les Graphes, Application aux Problèmes d'Ordonnancement, Metra Spécial Série, No. 1, Paris, 1962.

[3] Schwartz, J., On Programming: An Interim Report on the SETL Project, Installment 2: The SETL Language, and Examples of its Use, Computer Science Dept., Courant Inst. Math. Sci., New York Univ., 1973.