SETL Newsletter # 127

K. Kennedy
March 15, 1974

Edge-Listing Data-Flow Algorithms

The problem of constructing data-flow information from the control-flow graph of a program has been studied by a number of investigators [1,2,3,4,5,6]. In [3] the author presented an algorithm which uses "Cocke-Allen interval" analysis to solve the problem of locating "live" variables in a program. Hecht and Ullman [4] proposed a tabular method which has been shown to require more bit-vector operations than the interval method on some graphs and fewer on others [5].

This newsletter proposes an entirely new tabular approach which is applicable to most global data-flow problems. After an initial processing expenditure, this method is optimal in terms of bit-vector operations.

Edge-Listings

We define an *edge-listing* to be a sequence

$\ell = (e_1, e_2, \ldots, e_m)$

of edges in the program flow graph, where some edges may be repeated, such that every simple path in the flow graph is a subsequence of ℓ. That is, if

$$(d_1, d_2, \ldots, d_n)$$

are the edges of a simple path in the flow graph, there exist indices

$$j_1, j_2, \ldots, j_n$$

such that $j_i < j_{i+1}$, $1 \leq i < n$, and $d_i = e_{j_i}$, $1 \leq i \leq n$.

Certainly an edge-listing exists, becauce if $f_1, \ldots, f_k$ are all the edges in the graph then

$$\ell = (\underbrace{f_1, \ldots, f_k}, \underbrace{f_1, \ldots, f_k}, \ldots, \underbrace{f_1, \ldots f_k})$$

with k repetitions of $(f_1, \ldots, f_k)$ is a valid edge-listing. An edge-listing is said to be *minimal* if there is no shorter edge-listing for the same graph.

Data Flow

Most data flow problems can be expressed by "edge-equations" on the edges of the control flow graph. For example, consider the problem of identifying "live" variables. The following sets are important.

1. *live(b)* - the set of variables which are live on entry to the block b - where a variable is "live" at a point if there exists a path from that point to a use of the variable which path contains no redefinition of the variable.

2. *inside(b)* - the set of variables for which there is a use not preceded by a definition in b.

3. *thru(b)* - the set of variables which are neither used nor defined in b.

It has been shown [3,5] that the following class of equations defines the problem:

$$\text{(1)} \quad \text{live}(b) = \text{inside}(b) \cup \bigcup_{k \in S(b)} (\text{thru}(b) \cap \text{live}(k))$$

where S(b) is the set of successors of the block b in the control flow graph.

The essence of the edge-listing method is to propagate the "live" information backwards along all simple paths by applying the following analog of equation (1)

(2) $\text{live}(b) = \text{live}(b) \cup (\text{thru}(b) \cap \text{live}(k))$

(*live(b)* is initially *inside(b)*) on each edge (b,k) of an edge-listing in reverse order. The following SETL algorithm does this. Its argument *edgelist* is an edge-listing represented as a tuple of pairs <b,k>.

```
definef liveanalysis (nodes, edgelist, thru, inside)
     /* initialize live to inside */
     ( ∀b ε nodes)  live(b) = inside(b); end ∀b;
     /* iterate through the edge-listing */
     (# edgelist ≥ ∀i ≥ 1 )   /* reverse order  */
          <b,k> = edgelist(i);
          live(b) = live(b) + (thru(b) * live(k));
     end ∀i;
     return live;
end liveanalysis;
```

This simple algorithm is optimal whenever the edge listing is minimal.

Our only remaining problem is to find an algorithm which generates a minimal edge listing.

Enumerating Simple Paths

The first step in the generation of a minimal edge listing is the enumeration of all simple paths in the graph. For each node n, we will compute *path(n)*, the set of simple paths which terminate at n. This computation can be performed by an iterative method similar to Kildall's [6].

Initially, *paths(n)* contains only the null tuple. We iterate through the nodes n of the graph adding to *paths(n)* all paths leading to a predecessor k of n and through k to n (which paths do not contain n). When no new paths are added during an iteration, we halt.

The following is a SETL function which accepts *graph*, a tuple of nodes, edges, and entry node, and produces the *paths* sets.

```
definef computepaths (graph);
     /* break graph into component parts */
     <nodes, edges, entry> = graph;
     /* compute the predecessor function */
     preds = {<tℓx, hd x> x ε edges} ;
     /* initialize the paths to null tuples */
     (∀n ε nodes) paths(n) = {nult} ; end ∀n;
     /* iterate through the graph until there
        are no changes */
     change = t;
     (while change)    change = f;
          /* recompute paths for each
             node in the graph */
          (∀n ε nodes)
              newpaths = [+: k ε preds {n}] paths(k);
              /* check each new path for the
                 occurrence of  n  */
              (∀ p ε newpaths)
                   if ( 1 ≤ ∀j ≤ # p | hd p(j) ne n)
                      and (p + <<k ,n>>) n ε paths(n)
                      then paths(n) = paths(n) with (p + <<k,n>>);
                           change = t; end if;
              end ∀p;
          end ∀n;
     end while;
     return paths;
end computepaths;
```

Our next task is to eliminate some duplication. We take all the paths into one set and eliminate those which are contiguous within another path.

```
definef reduce (paths);
     /* pool all paths */
     allpaths =  [+: x ε paths ] tℓ x ;
     /* check for contiguous subsequences */
     (∀t ε allpaths |∃(  q ε allpaths - {t}, 1 ≤  ∃j ≤ (#q-#t+1)
                                         | q(j: # t) eq t ))
          allpaths = allpaths - {t};
     end ∀t;
  return allpaths;
end reduce;
```

On completion of this function, we are left with a set of maximal simple paths.

<u>Edge-Listing Generation</u>

We must now find an edge sequence which contains each of the enumerated paths as a subsequence. To do this we will use an exhustive search through a tree of possibilities.

We build a tree in which we have two quantities associated with each node:

(1) *sofar(node)* - the partial edge-listing generated so far.

(2) *rpaths(node)* - the set of paths remaining to be accounted for.

In other words, each node represents a sequence of decisions which have led to the partial listing *sofar(node)*. When a new node is created by a decision to add a certian edge to the listing, that edge is stripped off the beginning of each remaining path to create a new *rpaths* set. When the set of remaining paths reduces to a null tuple, the edge-listing is complete. Each leaf in the completed tree represents a valid edge-listing, the shortest being a minimal edge-listing.

The SETL function *multimerge* uses an adaptation of this method. The tree is built in stages: first all possible initial sequences of length 1 are computed, then those of length 2, and so on. We stop whenever all paths have been exhausted for some leaf because when this happens we will have a minimal listing. A small speed-up is attained by the following trick: Whenever an edge appears only at the beginning of the remaining paths we can add this edge to the partial sequence without considering other possible decisions - thus we can move to the next stage with only one son tending from this node.

The following SETL function creates a new tree node, given the sets *sofar* and *rpaths* for the parent node (*seq* and *rempaths*, respectively) and the selected next edge *x*. If, after stripping *x* from the remaining paths, all paths have been accounted for, the edge listing is returned as the value; otherwise, the new node is added to the argument set *new*.

```
definef createnode ( x, seq, rempaths, new);
     /* get a new atom */
     node = newat;
     sofar(node) = seq + < x >;
```

```
      /* strip initial edges from remaining paths to
         create a new rpaths set */
      rpaths(node) = nℓ;
     (∀t ε  remset)
         if  x  eq  t(1)
             then rpaths(node) = rpaths(node)  with (tℓ t);
             else rpaths(node) = rpaths(node) with t;
         end if x;
      end ∀t;
      /* test to see if remaining paths are null */
      if rpaths(node) eq  {<nult}
         then return (sofar(node));
         else new = new with node;
                   return nult;
         end. if;
end createnode;
```

Finally we present the routine *multimerge* which builds the tree. Note that the operator seqelt returns the index of an element in a tuple.

```
definef multimerge (pathset)
     /* initialize the first node */
     node = newat;
     rpaths(node) = pathset;
     sofar(node) = nult;
     nodes =  {node} ;
     (while nodes  ne  nℓ doing nodes = new; new = nℓ;)
          /* iterate through the tree nodes at this stage */
          (∀n ε nodes)
              <seq, remset> = <sofar(n), rpaths(n)>
              /* check for special condition */
              if   ∃t ε  remset | (∀q ε remset |(t(1)seqelt q)ℓe 1)
                 then  /* create only one son */
```

```
                test = createnode (t(1), seq, remset, new);
                if test ne nult then return test;;
            else /* multiple branches */
               startset = hd [remset];
               (∀x ε startset)
               test = createnode(x, seq, remset, new);
               if test ne nult then return test;;
               end ∀x;
         end if;
      end ∀n;
   end while;
end multimerge;
```

The operator <u>seqelt</u> is programmed as follows.

```
definef a seqelt  t;
    return (if 1 ≤  ∃i ≤ # t | t(i)eq a then i else 0);
end seqelt;
```

The method we have presented is clearly "brute force". It is probably not worth the effort to compute an edge-listing with such a time-consuming method. However, it does provide us with a method of evaluating heuristic edge-listing generators.

Summary

We have introduced the concept of a *minimal edge-listing* and we have used such a listing to rapidly compute global data-flow information. A time-consuming algorithm to generate a minimal edge-listing has been presented. This algorithm will be used in the evaluation of heuristic generators to be discussed in future newsletters.

Acknowledgment. This work was supported by the National Science Foundation, Grant NSF - OCA - GJ - 40585.

References

1. Cocke, J. "Global Common Subexpression Elimination", SIGPLAN NOTICES, Vol. 5, No. 7, pp. 20-24, July 1970.

2. Allen, F.E., "A Basis for Program Optimization", PROC. IFIP Conf. 71, North Holland Publishing Co. Amsterdam, 1971.

3. Kennedy, K., "A Global Flow Analysis Algorithm", International J. Computer Math., Section A, Vol. 3, pp. 5-15, Dec. 1971.

4. Hecht,M.S., and Ullman,J.D., "Analysis of a Simple Algorithm for Global Data Flow Problems", Proc. of ACM Conference on Principles of Programming Languages, Boston, Mass. October 1973.

5. Kennedy,K., "A Comparison of Algorithms for Global Flow Analysis", Technical Report 476-093-1, Dept. of Mathematical Sciences, Rice University, Houston, February 1974.

6. Kildall, G.A., "A Unified Approach to Global Program Optimization", Proc. of ACM Conference on Principles of Programming Languages, Boston, Mass. October 1973.