Acta Informatica 24, 679–694 (1987)

An Efficient General Iterative Algorithm for Dataflow Analysis

Susan Horwitz[1], Alan Demers[2], and Tim Teitelbaum[3]

[1] Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, USA
[2] Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
[3] Department of Computer Science, 405 Upson Hall, Cornell University, Ithaca, NY 14853, USA

Summary. Existing iterative algorithms for global dataflow analysis have demonstrable shortcomings; either they can be used only for a limited class of problems or they are needlessly inefficient in some cases. We review several algorithms, pointing out weaknesses and develop a new algorithm that can be used for a wide class of problems and has a runtime that compares favorably ro runtimes of existing algorithms.

1. Introduction

A number of iterative methods for performing global dataflow analysis on a flowgraph representation of a program have been proposed. The most efficient, node listing iteration [1, 9], has worst-case runtime $O(n \log n)$, but can be applied only to a limited class of flow problems. For more general iterative algorithms, worst-case runtimes depend on the particular problem being solved; for many problems, including those in the domain of node listing iteration, worst-case runtime is $O(n^2)$.

Non-iterative algorithms can also be used for global dataflow analysis. Two important non-iterative methods are presented in [5] and [13, 14]. Although they may appear to out-perform the iterative algorithms, there is an important difference in the way runtimes are calculated for these methods and for iterative methods. The two non-iterative methods require function compositions; applications of both composed and non-composed functions are given equal weight in the runtime analyses. By contrast, iterative algorithms apply *only* non-composed functions.

The functions associated with certain specific dataflow problems, e.g., live variable analysis, *can* be composed so that an application of a composed function is equivalent to an application of a non-composed function in terms of runtime. It is not clear, however, that such efficient compositions exist for *arbitrary* functions. It is, therefore, misleading to compare the time requirements of the two non-iterative algorithms with those of general iterative algorithms. The non-iterative algorithms do not necessarily supercede the iterative approach in the context of *arbitrary* dataflow problems.

We confine our discussion to iterative methods for the remainder of the paper. Our goal is to develop a general iterative algorithm with the following properties:

(1) It is as efficient as node listing iteration when applied to a problem in the domain of node listing iteration.

(2) It is never less efficient than existing general iterative algorithms.

(3) There exist examples on which it is more efficient than all existing general iterative algorithms.

The structure of the rest of the paper is as follows: Sect. 2 introduces terminology; Sect. 3 discusses the two existing general iterative algorithms, *worklist iteration* [11], and rPOSTORDER *iteration* [6, 8], including examples on which they may require $O(n^2)$ time when $O(n)$ time is clearly sufficient. Section 3.2 develops a series of new iterative algorithms; the final version, presented in Sect. 3.2.4 has the three properties listed above. Knowledge of node listing, worklist, and rPOSTORDER iteration is assumed.

2. Terminology

For the data flow analysis algorithms discussed here, a program is represented by a flowgraph $G=(N, E, n_0)$, where (N, E) is a directed graph, and n_0 in N is the entry node. Nodes in N represent basic blocks of the program; edges in E represent possible transfer of control among basic blocks. We assume that there is a path from n_0 to every node in N. For edge $e=(n_1, n_2)$ we define $\text{source}(e)=n_1$ and $\text{target}(e)=n_2$.

For programs written using standard constructs (including GOTO*s*), the number of edges in the flowgraph is O(*number of nodes in the flowgraph*). We use n to denote the number of nodes (and edges) in the flowgraph when discussing runtimes.

A flow problem is either a *forward* problem or a *backward* problem. Given a point in a program, forward problems determine what could happen before program execution reaches that point; backward problems determine what could happen during or after execution at that point. Available-expression analysis and constant propagation are examples of forward problems; live-variable analysis and faint-variable analysis [4] (see Appendix A) are examples of backward problems. To solve backward problems, some algorithms require that the flowgraph have a unique exit node; if the flowgraph does not have such a node, one can be added, with an incoming edge from every exit node of the original graph. All the algorithms discussed here can handle both forward and backward problems, so we will not differentiate between them; in particular, algorithms that include schemes for numbering the nodes of a flowgraph are always given as if for forward problems; backward problems simply use the reverse numbering.

The goal of dataflow analysis is to produce an *assignment*, i.e. to assign to each node in N a *program fact*, information that will be valid every time the node is reached during every possible execution. The universe of program facts is modeled by a bounded (contains no infinite descending chains) lower semi-lattice L with meet operation $\wedge$, least element $\perp$ (bottom), and partial order $\leqq$. Some dataflow algorithms also require a greatest element $\top$ (top); if L does not include such an element, a new value $\top$ not an element of L can be added, defined so that for all x in L, $x \wedge \top = x$.

Associated with each edge e in E is a function f_e: $L \to L$, such that if fact x is true before executing source(e), and control flows along e, then fact $f_e(x)$ will be true before executing target(e).

Definition. A function f: $L \to L$ is *monotone* iff for all x, y in L, $x \leqq y$ implies $f(x) \leqq f(y)$.

Definition. A *monotone dataflow framework*[1] consists of:

(1) a bounded lower semi-lattice L with meet operation $\wedge$, least element $\perp$, and partial order $\leqq$;

(2) a value n_0_init, an element of the set $\{\perp, \top\}$; n_0_init represents the program fact initially true at entry node n_0;

(3) a set F of monotone, total functions from L to L closed under composition; further, we require that for every element x in L there exists a function f in F such that $f(n_0\text{\_init}) = x$.

Definition. An *instance* of a dataflow problem consists of:

(1) a monotone dataflow framework $(L, F, n_0\text{\_init})$;

(2) a flowgraph $G = (N, E, n_0)$;

(3) a map M: $E \to F$.

Map M associates a function from F with each edge of the flowgraph; we use f_e to denote $M(e)$.

We extend our notation to include f_p, the function associated with path P, as follows: if P is the empty path then $f_p(x) = x$; if $P = (e_1, e_2, \ldots, e_k)$, then $f_p(x) = f_{e_k} f_{e_{k-1}} \cdots f_{e_1}(x)$.

Given an instance of a dataflow problem, the ideal result of dataflow analysis is to produce the *meet-over-all-paths* assignment, the map MOP: $N \to L$ such that for all nodes n, $\text{MOP}(n) = \wedge \{f_p(n_0\text{\_init}) \mid P \text{ is a path from } n_0 \text{ to } n\}$. Unfortunately, the MOP assignment is, in general, undecidable [7]. Fortunately, there are assignments other than the MOP that are both decidable and useful; one such is defined below.

Definition. A *fixed point assignment* for an instance of a dataflow problem is a map FP: $N \to L$ such that for all edges e, $\text{FP}(\text{target}(e)) \leqq f_e(\text{FP}(\text{source}(e)))$.

Theorem [7]. *For every instance of a dataflow problem there exists a unique maximum fixed point assignment* MFP, *and for all nodes n,* $\text{MFP}(n) \leqq \text{MOP}(n)$.

[1] Other frameworks that have been studied include *distributive* [7, 8, 11], *continuous* [13], and *k-bounded* [5, 13]

Some dataflow analysis algorithms [10, 15] produce assignments A that may be less precise than the maximum fixed point assignment MFP. By less precise we mean that there may be node n such that $A(n) < \mathrm{MFP}(n)$. All algorithms considered in this paper produce the MFP assignment for instances of dataflow problems.

Some algorithms guarantee correctness only for limited classes of flow problems. These algorithms may restrict the flowgraph G, the set of functions F, or both G and F. Node listing iteration requires that the flowgraph be *reducible* [3] and that the functions be *separable* [6].

Definition. A dataflow framework is *separable* iff for all f, g in F, x in L:

$$fg(n_0\_\mathrm{init}) \geqq g(n_0\_\mathrm{init}) \wedge f(x) \wedge x.$$

Separability is an important concept in this paper; our goal will be to develop an algorithm that works efficiently on both separable and non-separable problems. Faint-variable analysis, defined and discussed in Appendix A, is an example of a non-separable but obviously useful flow problem. Another example commonly found in the literature is constant propagation.

3. Iterative Algorithms

We now turn our attention to iterative methods for solving dataflow problems. Our goal is to produce a new general iterative algorithm that outperforms existing algorithms. In Sect. 3.1 we look at examples that may cause the two existing general iterative algorithms, worklist iteration and rPOSTORDER iteration, to exhibit pathological behavior; in Sect. 3.2 we develop a series of new algorithms, each of which attempts to overcome some problem found in a previous version. Our final algorithm, presented in Sect. 3.2.4, is as efficient as node listing iteration when applied to a problem in the domain of node listing iteration, is never worse than worklist or rPOSTORDER iteration, and, on some examples, is shown to be better than both worklist and rPOSTORDER iteration.

We wish to stress that the examples presented in the following section were designed specifically to cause worklist and rPOSTORDER iteration to exhibit worst-case behavior. We make no claims about the probability of such examples resulting from "real-life" programs and thus do not claim that these algorithms will exhibit worst-case behavior when used on "real-life" programs.

3.1. Old Iterative Algorithms

All Iterative algorithms are based on the following non-deterministic algorithm, which we call the *canonical iterative algorithm.*

Input: an instance of a dataflow problem
Output: for each node n of flowgraph G, the value $n.\mathrm{val} = \mathrm{MFP}(n)$

```
LET "visit $n$" be defined as:
    LET $e$ be any edge such that target($e$)$=n$ IN
        **begin**
        temp$:=f_e$(source($e$).val);
        **if** temp$<n$.val **then** $n$.val$:=$temp **fi**
        **end**
IN
**begin**
    $n_0$.val$:=n_{0\_}$init;
    **for** all other nodes $n$ **do** $n$.val$:=$top **od**;
    **while** there exists a node $m=$target($e$) such that $m$.val
        $>f_e$(source($e$).val) **do**
        choose any node $n$;
        visit $n$
        **od**
    **end**
```

Fig. 1. The canonical iterative algorithm

Note that the canonical iterative algorithm may never terminate because the node n that is chosen in the body of the **while** loop may not be one of the nodes m with an inconsistent value mentioned in the loop condition. The algorithms discussed below use different methods to insure termination.

3.1.1. Worklist Iteration. During worklist iteration, n, the next node to be visited, is chosen from a worklist, which is initialized to contain all nodes; if n.val changes as a result of the visit, all successors of n are added to the worklist. Worklist iteration can be used to find the MFP assignment for both separable and non-separable flow problems, on both reducible and non-reducible flowgraphs. Worst-case runtime is $O(n *$ *length of longest chain in lattice*$)$.

The problem with worklist iteration is that there is no specified order for choosing nodes from the worklist. For example, consider the graph shown in Fig. 2:

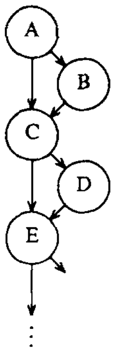

Fig. 2. Example flowgraph I

The topological order of the nodes is: ABCD... It is possible to choose nodes from the worklist in topological order, and this results in $O(n)$ runtime. It is also possible to choose nodes so as to respect the *reverse* topological order:

after visiting node A, both B and C will be placed on the worklist; choose C; after visiting C, BD and E will be on the worklist; choose E; and continue in this manner, always choosing the node closest to the end of the alphabet.

For certain problems, such as available expression analysis, the length of the longest chain in the lattice will be $O(n)$; in that case, the worst-case choosing strategy described above produces $O(n^2)$ runtime, when $O(n)$ is clearly sufficient.

3.1.2. rPOSTORDER Iteration. rPOSTORDER iteration consists of multiple passes through the flowgraph, on each of which all nodes are visited in reverse postorder; the algorithm halts when no n.val changes during a pass. rPOSTORDER iteration is as general a technique as worklist iteration, and will achieve $O(n)$ runtime on flowgraphs like that of Fig. 2; however, the worst-case runtime is $O(n * \textit{length of longest chain in lattice})$, just as for worklist iteration.

We can take advantage of the fact that rPOSTORDER iteration visits every node of the graph on each iteration to produce an example on which rPOSTORDER iteration is unnecessarily slow. Consider the flowgraph shown in Fig. 3.

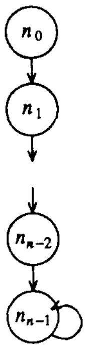

Fig. 3. Example flowgraph II

All nodes except n_{n-1} will receive final values during the first iteration; for a non-separable problem, the number of iterations required to produce the final value for node n_{n-1} could be equal to the length of the longest chain in the lattice. In that case, runtime for rPOSTORDER iteration would be $O(n * \textit{length of longest chain})$, while an algorithm that visited only node n_{n-1} after the first iteration would have runtime $O(n + \textit{length of longest chain})$.

3.2. New Iterative Algorithms

In this section we explore ways of modifying and combining existing algorithms. We wish to produce a new algorithm that is as general as worklist or rPOSTORDER iteration but avoids the possibility of the kinds of pathological behavior discussed above in Sect. 3.1. Our new algorithms are still based on the canonical iterative algorithm; in particular, "visit *n*" is defined as in Fig. 1.

3.2.1. Priority-Queue Iteration. One nice feature of worklist iteration is that it visits a node only when some immediate predecessor has changed, unlike rPOST-ORDER iteration, which visits every node on every iteration. The problem with worklist iteration is that it includes an unspecified "choose" operation: choose a node from the worklist; thus, in analyzing the worst-case runtime of worklist iteration we must always assume the worst-case choice.

Suppose that, instead of keeping an unordered worklist, we keep a priority queue [2], with nodes ordered in reverse postorder. We will call this *priority-queue iteration.* At first, priority-queue iteration seems to combine the best features of worklist and rPOSTORDER iteration. Because it is the same as worklist iteration but with a specified choose operation, worst-case runtime (in terms of the number of function applications) can never be worse than for worklist iteration, and there are obvious examples (such as Fig. 2) where priority-queue iteration will be much better than worklist iteration. Maintaining the priority queue is not free, but it takes only $\log n$ (where n is the size of the queue, at most the number of nodes in the flowgraph) time for each insert or delete operation. In the worst case, total runtime for priority-queue iteration is only a factor of $\log n$ worse than for worklist iteration; in the best case, it is a factor of (*length of longest chain*) better; in Fig. 2 for example, worklist iteration may take $O(n^2)$ time, while priority-queue iteration takes just $O(n)$ time because the size of the priority queue never exceeds a small constant.

Priority-queue iteration can be an improvement on rPOSTORDER iteration, too, for non-separable problems. The flowgraph given in Fig. 3, which could require $O(n * \textit{length of longest chain})$ function applications for rPOSTORDER iteration, would require just $O(n + \textit{length of longest chain})$ applications for priority-queue iteration.

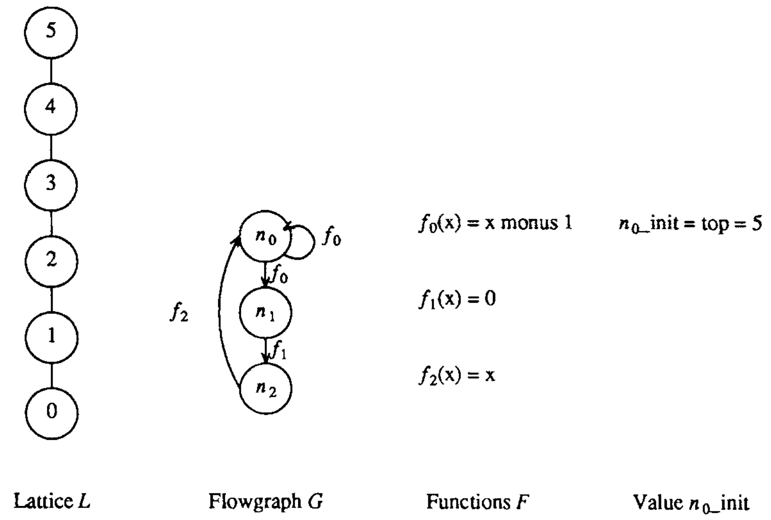

Fig. 4. Example dataflow problem instance

Unfortunately, priority-queue iteration can be arbitrarily worse than rPOSTORDER iteration. Figure 4 gives an instance of a non-separable flow problem for which rPOSTORDER iteration requires just $O(n)$ time, using three iterations: one to initialize, one to propagate values, and one to discover that final values have been assigned to all nodes. Priority-queue iteration requires O(*length of longest chain*) time; because n_0 is a successor of itself, n_0 will always be placed at the front of the priority queue until successive applications of f_0 to n_0_init finally produce bottom. Theoretically, the length of the longest chain in the lattice can be arbitrarily greater than $O(n)$, so the runtime of priority queue iteration can be arbitrarily worse than that of rPOSTORDER iteration.

What is it about this example that causes priority-queue iteration to have such bad behavior compared to rPOSTORDER iteration? The flowgraph of Fig. 4 is strongly connected, so that a change in the value at any node may influence the values at all other nodes in the graph; however, priority-queue iteration visits one node, n_0, many times before visiting other nodes at all. By contrast, rPOSTORDER iteration visits every node in a strongly connected component once during each iteration.

3.2.2. Strongly-Connected-Component Iteration. The observations at the end of the previous section suggest a new approach, which we call *strongly-connected-component iteration (scc iteration)*: visit strongly connected components in topological order, using rPOSTORDER iteration *within* each scc. An algorithm for scc iteration is given below.

Input: an instance of a dataflow problem
Output: for every node n of flowgraph G, the value n.val = MFP(n)

Step 1: Numbering the graph

```
LET D be the dag produced by finding all strongly connected
    components in G and reducing each to a single node
IN
begin
    num:=0;
    for each node d in D in topological order do
        for each node n in the scc represented by d in
            reverse postorder do
            n.number:=num;
            num:=num+1
            od
        od
    end
```

Reverse postorder within each scc is obtained using a depth-first search that starts with the entry node n_0, if n_0 is within this scc, and otherwise starts with any node that is the target of an edge whose source is in a previous scc.

```
Step 2: Flow analysis
        begin
           **initialize**
              $n_0.val := n_0$_init;
              for all other nodes $n$ do
                 $n$.val := top
                 od;
           **find MFP**
              for each scc in topological order do
                 perform rPOSTORDER iteration within this scc, i.e.
                 using $n$.number, visit all nodes $n$ in this scc in
                 reverse postorder, repeat until no $n$.val changes
                 od
           end
```

Step 1 can be done in $O(n)$ time [2]. We show below that the time needed for Step 2 is never more than the time needed for flow analysis using rPOSTORDER iteration.

Our proof is based on the fact that an application of either rPOSTORDER or scc iteration can be represented by a *visit sequence*, a list of the nodes as they are visited during the application. The length of the visit sequence corresponds to the runtime of the algorithm; thus, we show that, for an arbitrary instance of a dataflow problem, the length of the visit sequence that represents the application of scc iteration to the problem is never longer than the length of the visit sequence that represents the application of rPOSTORDER iteration to the problem.

An application or rPOSTORDER iteration consists of m iterations, each of which visits all nodes in reverse postorder; when the application is finished, each node has been assigned a final value. An application can be represented by listing the nodes in reverse postorder m times to form a visit sequence. For example, rPOSTORDER iteration might require three iterations to produce the MFP assignment for a flow problem using the flowgraph of Fig. 5.

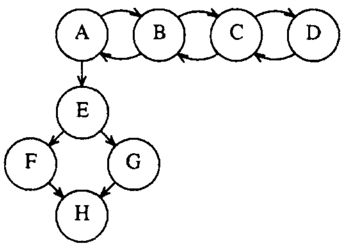

Fig. 5. Example flowgraph III

Such an application would be represented by the following visit sequence:

ABCDEFGHABCDEFGHABCDEFGH

Similarly, the application of scc iteration to the same problem would be represented by the following visit sequence:

ABCDABCDABCDEFGH

In general, the visit sequence representing an application of rPOSTORDER iteration to a flowgraph with k strongly connected components that requires m iterations looks like:

$$\overbrace{(\mathrm{scc}_1)\,(\mathrm{scc}_2)\ldots(\mathrm{scc}_k)}^{1}\,\overbrace{(\mathrm{scc}_1)\,(\mathrm{scc}_2)\ldots(\mathrm{scc}_k)}^{2}\ldots\overbrace{(\mathrm{scc}_1)\,(\mathrm{scc}_2)\ldots(\mathrm{scc}_k)}^{m}.$$

The visit sequence representing an application of scc iteration looks like:

$$(\mathrm{scc}_1)\,(\mathrm{scc}_1)\ldots(\mathrm{scc}_1)\,(\mathrm{scc}_2)\,(\mathrm{scc}_2)\ldots(\mathrm{scc}_2)\ldots(\mathrm{scc}_k)\,(\mathrm{scc}_k)\ldots(\mathrm{scc}_k).$$

It is clear that the final value at node n can depend on values at node m only if there is a path from m to n. This observation motivates the following definitions and rule:

Definition. Two visit sequences VS1 and VS2 are *equivalent* iff visiting nodes according to VS1 results in the same final values at all nodes as visiting nodes according to VS2.

Definition. For a given flowgraph G, node x is an *ancestor* of node y iff there exists a path in G from x to y.

Definition. A *group* of nodes is a non-empty consecutive sub-sequence of a visit sequence.

Rule. If visit sequence VS contains two consecutive groups of nodes A and B such that no node in A is an ancestor of a node in B, then VS is equivalent to the visit sequence identical to VS except with AB replaced by BA.

Example. The visit sequence given above for rPOSTORDER iteration applied to the flowgraph of Fig. 5 contains consecutive groups (EFGH) and (ABCD); no node in the first group is an ancestor of a node in the second group, thus, the above visit sequence is equivalent to the following (with the changed part in italics):

ABCD*ABCDEFGH*EFGHABCDEFGH

Theorem. *An application of scc iteration will visit no more nodes than an application of* rPOSTORDER *iteration.*

Proof. It is sufficient to show that, for an arbitrary instance of a dataflow problem, the visit sequence V1 representing the application of scc iteration to the problem is never longer than the visit sequence V2 representing the application of rPOSTORDER iteration to the same problem. For $i<j$, no node in scc_j can be an ancestor of a node in scc_i; thus, using the rule given above we can permute V2 to the equivalent form:

$$\overbrace{(\mathrm{scc}_1)}^{1}\,\overbrace{(\mathrm{scc}_1)}^{2}\ldots\overbrace{(\mathrm{scc}_1)}^{m}\,\overbrace{(\mathrm{scc}_k)}^{1}\,\overbrace{(\mathrm{scc}_k)}^{2}\ldots\overbrace{(\mathrm{scc}_k)}^{m}$$

which contains exactly as many nodes as the original form. Because this new form is equivalent to the old form, we see that, for all i, m passes through scc_i are sufficient to produce final values for all nodes in scc_i. An application of scc iteration produces the same final values as does an application of rPOSTORDER iteration, and an application of scc iteration visits scc_i only until all nodes in scc_i have received their final values; thus, there will never be more than m occurrences of scc_i in V1, and the length of V1 will be no greater than the length of V2.

3.2.3. Priority-Scc Iteration. Scc iteration is guaranteed to perform no more function applications than rPOSTORDER iteration, and it will perform fewer function applications on problem instances like those of Fig. 3. However, if we were to modify that flowgraph slightly by adding an edge from node n_{n-1} back to node n_0 then the entire graph would be one scc, and scc iteration would be no better than rPOSTORDER iteration.

What we really want is scc iteration where we only visit a node after a predecessor has changed. This idea can be expressed as follows:

```
number nodes as for scc iteration;
initialize graph as for scc iteration;
mark all nodes;
for each scc in topological order do
    while there is a marked node in this scc do
        for each marked node n in this scc in reverse postorder do
            unmark n;
            visit n;
            if n.val changed then mark all successors fi
            od
        od
    od
```

We can use a priority queue to implement the "mark" and "unmark" operations as we did for priority-queue iteration; a node is marked if it is in the queue. We can achieve the condition of the inner **for** loop, "... in reverse postorder", by using n.number to order the priority queue. To insure that the algorithm produces final values for one scc before going on to the next scc, and to insure that it visits all nodes within the scc once before visiting any nodes twice, we use three *concatenable* priority queues [2]:

(1) *future queue* contains all marked nodes in scc's with topological number greater than the current scc;

(2) *current_queue* contains all marked nodes in the current scc with reverse postorder number greater than the current node; and

(3) *pending_queue* contains all marked nodes in the current scc with reverse postorder number less than the current node.

We call this priority-scc iteration; an algorithm is given below.

```
LET
    number be defined as for scc iteration;
    scc's be numbered starting with 1;
    max(i) be the highest numbered node in scc_i;
    current_queue, pending_queue, and future_queue be concatenable prior-
        ity queues containing nodes n, ordered by n.number, with the follow-
        ing operations:
        EMPTY() return an empty queue
        NEXT(queue) remove and return the lowest numbered node from
            the given queue
        ADD(node, queue) add the given node to the given queue
        CAT(queue1, queue2) return queue1 concatenated to the front of
            queue2
        SPLIT(queue, number) remove and return (as a priority queue) all
                             nodes n in the given queue with reverse post-
                             order number less than or equal to the given
                             number
IN
begin
    current_queue:=EMPTY();
    pending_queue:=EMPTY();
    future_queue:=EMPTY();
    n_0.val:=n_0_init;
    for all other nodes n do n.val:=top od;
    for all nodes n do ADD (n, future_queue) od;
    current_scc:=0;
    while there exists a non-empty queue do
        if current_queue is empty
            then if pending_queue is non-empty
                    then current_queue
                    :=CAT(pending_queue, current_queue);
                        pending_queue:=EMPTY()
                    else current_scc:=current_scc+1;
                        current_queue:=SPLIT(future max(current_scc))
                    fi
            fi;
        n:=NEXT(current_queue);
        visit n;
        if n.val changed
            then for all nodes m, successors of n do
                    if m.number ≦ n.number
                        then ADD(m, pending_queue)
                        else if m.number ≦ max(current_scc)
                                then ADD(m, current_queue)
                                fi
                        fi
                    od
            fi
        od
    end
```

The only difference between priority-scc iteration and scc iteration is that priority-scc iteration never visits a node with unchanged predecessors; thus, priority-scc iteration is at least as fast (in terms of the number of function applications) as scc iteration and rPOSTORDER iteration. Priority-scc iteration differs from priority iteration only by refusing to consider any node in an scc twice until all nodes in that scc with changed predecessors have been considered once; thus, priority-scc iteration is also a special case of worklist iteration and it will be at least as fast as worklist iteration, again in terms of the number of function applications. Bookkeeping costs for maintaining a concatenable priority queue are the same, $O(\log(\textit{size of the queue}))$ as for a priority queue and $O(\log n)$ in the worst case.

Figure 6 gives an example on which priority-scc iteration will perform better than both rPOSTORDER and worklist iteration, even counting the extra time needed for bookkeeping.

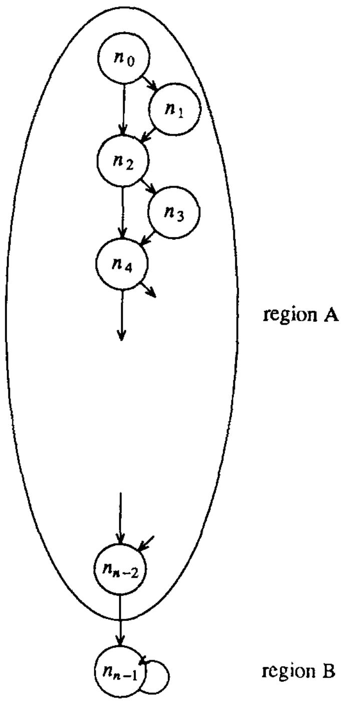

Fig. 6. Example flowgraph IV

In Fig. 6, region A is the same as the flowgraph of Fig. 2, so if the length of the longest chain in the lattice is $O(n)$, worklist iteration may take $O(n^2)$ time to produce final values for the nodes in this region. The general form of the flowgraph of Fig. 6 is the same as that of Fig. 3, so rPOSTORDER iteration may require $O(n^2)$ time to produce final values for all nodes. Priority-scc iteration will visit the nodes of region A in topological order, taking time $O(n)$ to produce their final values, then will visit the node of region B $O(\textit{length of}$

longest chain) times (i.e. $O(n)$ times) to produce its final value. Total time for priority-scc iteration will be $O(n)$ including bookkeeping time because the priority queue will never contain more than two nodes.

3.2.4. Hybrid Iteration. Although priority-scc iteration is more general than node listing iteration, there are obvious examples using instances of separable flow problems where it can be much less efficient; node listings are never longer than $O(n \log n)$ while priority-scc iteration can require $O(n^2)$ function applications. We would like to guarantee $O(n \log n)$ runtime for separable problems and reducible flowgraphs, while maintaining the generality of our algorithm. A strategy like:

```
given an instance of a flow problem
if problem is separable and graph is reducible
    then apply node listing iteration
    else apply priority-scc iteration
    fi
```

would work if we know how to implement the separability test. Fortunately, such a test is not necessary; we achieve essentially the same result with the algorithm given below.

```
if the given flowgraph is non-reducible
    then perform priority-scc iteration
    else

Step 1: Apply node listing iteration;
Step 2: Apply priority-scc iteration using the values produced by Step 1 to ini-
        tialize all n.val.
```

We call this *hybrid iteration.*

It is easy to verify that hybrid iteration will produce the MFP assignment for both separable and non-separable problems:

– When applied to a separable problem, Step 1, node listing iteration, will produce the MFP assignment; Step 2 will visit each node once, producing no changes, and so the algorithm will halt.

– When applied to a non-separable problem, Step 1 will produce values $n.\mathrm{val}$ such that for all n, $n.\mathrm{val} \geq \mathrm{MFP}(n)$. Priority-scc iteration will produce the MFP assignment, given such initial values as long as its priority queue is initialized to include all nodes n such that $n.\mathrm{val} > \mathrm{MFP}(n)$. Priority-scc iteration initializes the priority queue to contain *all* nodes, thus, Step 2 will halt when and only when the MFP assignment has been produced.

There are examples on which hybrid iteration may perform more function applications than worklist or rPOSTORDER iteration because some flowgraphs require node listings of length $O(n \log n)$. Step 1 of hybrid iteration will perform $O(n \log n)$ function applications in such cases even though $O(n)$ function applications may be sufficient. To insure that the number of function applications performed by hybrid iteration is never more than the number performed by priority-scc iteration we modify the algorithm slightly by introducing parallelism:

```
In parallel, on separate copies of the flowgraph, do:
Process 1: Perform hybrid iteration as defined above.
           Kill process 2.
Process 2: Perform priority-scc iteration.
           Kill process 1.
```

Both processes will eventually produce the MFP assignment; parallelism allows us always to use the faster of the two.

4. Conclusion

The parallel version of hybrid iteration, presented above, achieves the goals listed in Sect. 1. However, these results are based on worst-case runtime analysis; empirical studies are needed to determine which is the best algorithm from a practical point of view. For example, [6] and [8] show that rPOSTORDER iteration, when applied to a separable problem, will have runtime at most $O(n*(d+3))$, where d is the maximum number of backedges in a cycle-free path through the flowgraph. In structured programs d is the maximum depth of loop nesting, shown in [12] usually to be three or less.

This result, which holds for priority-scc iteration as well, may mean that, in practice, conceptually simple algorithms like rPOSTORDER and priority-scc iteration are preferable to theoretically faster but conceptually more difficult algorithms like node listing and hybrid iteration.

Appendix A

Faint Variable Analysis

For live variable analysis, a variable x is defined to be live at a program point if there exists a path from that point to the end of the program on which x is used before being defined. A variable that is not live is dead. This idea was extended in [4] as flows:

> A variable x is defined to be *faint* at a program point if x is dead at that point or if x is live only because it is used to define a faint variable.

Faint variable analysis seeks to identify the minimal set of non-faint variables at each program point.

Live variable analysis can be used to find *useless assignments*, that is, assignments to dead variables. Removing such assignments during compilation will lower a program's space and time consumptions. The removal of a useless assignment may cause a previously live variable to become dead; thus, the use of live variable analysis to remove all useless assignments requires repeated applications of live variable analysis alternating with the removal of useless assignments.

Alternatively, faint variable analysis can be used to find all useless assignments in a single pass; an assignment is considered useless if the variable assigned to is faint, and removal of such an assignment cannot create new faint variables. In addition, faint variable analysis can find some useless assignments that even repeated live variable analysis will never find, as illustrated below:

```
x:=0;
while cond do
        (no use of x in this region)
    x:=x+1;
        (no use of x in this region)
    od;
        (no use of x in remainder of program)
```

The variable x is faint throughout this program because it is used only to define a faint variable (itself), and thus, both assignments to x could be removed. Live variable analysis would not discover this, because x is live throughout the loop.

References

1. Aho, A., Ullman, J.: Node listings for reducible flowgraphs. J. Comput. Syst. Sci. **13**, 286–299 (1976)
2. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Reading, MA: Addison Wesley 1974
3. Allen, F.: Control flow analysis. SIGPLAN Notices **5**, 1–19 (1970)
4. Giegerich, R., Moncke, U., Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. Proceedings, Third Conference of the European Co-operation in Informatics. In: Informatik-Fachberichte 50, pp. 1–10. Berlin Heidelberg New York: Springer 1981
5. Graham, S., Wegman, M.: A fast and usually linear algorithm for global flow analysis. J. ACM **23**, 172–202 (1976)
6. Hecht, M., Ullman, J.: Analysis of a simple algorithm for global data flow problems. Conf. Record of the 1st ACM Symp. on POPL 207–217 (1973)
7. Kam, J., Ullman, J.: Monotone data flow analysis frameworks. Acta Inf. **7**, 305–317 (1977)
8. Kam, J., Ullman, J.: Global data flow analysis and iterative algorithms. J. ACM **23**, 158–171 (1976)
9. Kennedy, K.: Node: Listings applied to data flow analysis. Conf. Record of the 2nd ACM Symp. on POPL 10–21 (1975)
10. Kennedy, K.: A survey of data flow analysis techniques. In: Program Flow Analysis, Theory and Applications. Muchnick, S., Jones, N. (eds.), pp. 45–46. Englewood Cliffs, NJ: Prentice Hall 1981
11. Kildall, G.: A unified approach to global program optimization. Conf. Record of the 1st ACM Symp. on POPL 194–206 (1973)
12. Knuth, D.: An empirical study of FORTRAN programs. Software Pract. Exp. **1**, 105–134 (1971)
13. Tarjan, R.: A unified approach to path problems. J. ACM **28**, 577–593 (1981)
14. Tarjan, R.: Fast algorithms for solving path problems. J. ACM **28**, 594–614 (1981)
15. Wilhelm, R.: Global flow analysis and optimization in the MUG2 compiler generating system. In: Program Flow Analysis, Theory and Applications. Muchnick, S., Jones, N. (eds.), pp. 144–147. Englewood Cliffs, NJ: Prentice Hall 1981

Received July 17, 1987/July 20, 1987