Two simple iterative algorithms for the available expressions problem

We will now use the available expressions problem to illustrate two simple iterative algorithms to compute the \( \text{in}[B] \) sets. We will use a matrix \( \text{in}[B,e] \), that will be one iff an expression \( e \) is in \( \text{in}[B] \). Similar matrices for \( \text{egen} \) and \( \text{ekill} \) will be used (but not necessarily implemented). To understand the algorithm, consider again the equation:

\[
\text{in}[B] = \bigcap_{C \in \text{PRED}[B]} \text{egen}[C] \cup (\text{in}[C] - \text{ekill}[C])
\]

We can rewrite it as a boolean equation:

\[
\text{in}[B, e] = \bigcap_{C \in \text{PRED}[B]} \text{egen}[C, e] \lor (\text{in}[C, e] \land \overline{\text{ekill}[C, e]})
\]

or

\[
\overline{\text{in}[B, e]}
\]

\[
= \bigcup_{C \in \text{PRED}[B]} ((\overline{\text{in}[C, e]} \land \overline{\text{egen}[C, e]}) \lor (\text{ekill}[C, e] \land \text{egen}[C, e]))
\]

Note here the \( \cap \) and the \( \cup \) symbols stand for big and and big or respectively.
begin
PILE = ∅
for each expression e in PROG do
    for each basic block B of PROG do
        if B is S or there is C in PRED(B) such that ekill[C,e]∧(¬egen[C,e]) then
            /* By second term of equation */
            in[B,e] = 0
            PILE += {B}
        else
            in[B,e]=1
        fi
    od
while PILE ≠ ∅ do
    C from PILE
    if ¬egen[C,e] then
        for each Bin SUC(C) do
            if in[B,e]=1 then
                /* By first term */
                in[B,e]=0
                PILE += {B}
            fi
        od
    fi
od
**Theorem.** The previous algorithm terminates and is correct.

**Proof**

*Termination.* For each expression $e$, no node is placed in PILE more than once and each iteration of the while loop removes an entry from PILE.

*Correctness.* $in[B,e]$ should be $0$ iff either:

1. There is a path from $S$ to $B$ such that $e$ is not generated in any node preceeding $B$ on this path or
2. there is a path to $B$ such that $e$ is killed in the first node ot this path and not subsequently generated.

If $in[B,e]$ is set to zero because of the second term of the equation clearly either 1 or 2 holds. A straightforward induction on the number of iterations of the while loop shows that if $in[B,e]$ is set to zero because of the first term of the equation, then 1 or 2 must hold.

Conversely if 1 or 2 holds, then a straightforward inuction shows that $in[B,e]$ is eventually set to $0$.

**Theorem** The algorithm requires at most $O(mr)$ elementary steps, where $m$ is the number of expressions and $r$ is the number of arcs.

**Proof** The “for each basic block” loop takes $O(n)$ steps. While loop considers each arc once. Therefore $O(r)$. Outermost loop is $m$ iterations and $r\geq n-1$. Therefore $O(mr)$
Here is another algorithm (Kildall’s):

\begin{verbatim}
begin
  in[1]=0^m
  in[2:n]=1^m
  for each C in SUC(1) do
    PILE += {(C,egen[1,:])}
  od
  for each node C do
    T(:) = egen[C,:] or \neg ekill[C,:]
    for each B in SUC(C) do
      PILE += {B,T}
    od
  od
  while PILE \neq \emptyset do
    (C,T) from PILE
    if (\neg T and in[C]) \neq 0^m then
      in[C,:] = in[C,:] and T(:)
      T(:)=(in[C,:] and \neg ekill[C,:]) or egen[C,:]
      PILE += {(D,T) | D in SUC(C)}
    fi
  od
end
\end{verbatim}
Round-Robin Version of the Algorithm

This is the most straightforward version:

begin
  \textit{in}[S] = \emptyset
  \textit{out}[S] = egen[S]
  \textbf{for each} block \( B \neq S \) \textbf{do}
    \textit{out}[B] = \textit{U} - ekill[B]
  \textbf{od}
  change = \textbf{true}
  \textbf{while} change \textbf{do}
    change = \textbf{false}
    \textbf{for each} block \( B \neq S \) /* in rPOSTORDER */
      \textit{in}[B] = \bigcap_{C \in \text{PRED}[B]} \textit{out}[C]
      oldout= \textit{out}[B]
      \textit{out}[B] = egen[B] \cup (\textit{in}[B] - ekill[B])
      \textbf{if} \textit{out}[B] \neq oldout \textbf{then}
        change = \textbf{true}
      \textbf{fi}
    \textbf{od}
  \textbf{od}
end
In bit notation

\begin{verbatim}
begin
  \text{in}[S, :] = 0^m
  \text{for each block } B \neq S \text{ do}
    \text{in}[B, :] = 1^m
  \text{od}
  \text{change} = \text{true}
  \text{while change do}
    \text{change} = \text{false}
    \text{for each block } B \neq S \text{ in rPOSTORDER do}
      \text{new} = \bigwedge C \in \text{PRED}[B] egen[C, :] \lor (\text{in}[C, :] \land \neg \text{ekill}[C, :])
      \text{if new} \neq \text{in}[B, :] \text{ then}
        \text{in}[B, :] = \text{new}
        \text{change} = \text{true}
      \text{fi}
    \text{od}
  \text{od}
end
\end{verbatim}
To study the complexity of the above algorithm we need the following definition.

**Definition** Loop-connectedness of a reducible flow graph is the largest number of back arcs on any cycle-free path of the graph.

**Lemma** Any cycle-free path in a reducible flow graph beginning with the initial node is monotonically increasing in rPOSTORDER.

**Lemma** The while loop in the above algorithm is executed at most $d+2$ times for a reducible flow graph, where $d$ is the loop connectedness of the graph.

**Proof** A 0 propagates from its point of origin (either from the source of from a “kill”) to the place where it is needed in $d+1$ iterations if it must propagate along a path $P$ of $d$ back arcs. One more iteration is needed to reach the tail of the first back arc.

**Theorem** If we ignore initialization, the previous algorithm takes at most $(d+2)(r+n)$ bit vector steps; that is $O(dr)$ or $O(r^2)$ bit vector steps.
Global Common Subexpression Elimination

A very simple algorithm is presented in the book by Aho Sethi and Ullman:

Algorithm GCSE: Global Common Subexpression Elimination

Input: A flow Graph with available expression information.

Output: A revised Flow graph

Method:

begin
    for each block B do
        for each statement s in B of the form x=y op z
            with y op z available at the beginning of B and
            neither y nor z defined before s in B do
                for each block C computing the expression
                y op z reaching B do
                    let t be the last statement in C of the
                    form w=y op z
                    replace t with the pair
                    u=y op z
                    w=u
                od
                Replace s with x = u
            od
        od
    end
To find the statements $t$, the algorithm searches backwards in the flow graph. We could compute something equivalent to use-definitions chains for expressions, but this may produce too much useless information.

Copy propagation can be applied to the program to eliminate the $w = u$ statement.

The algorithm can be applied several times to remove complex redundant expressions.
Copy propagation

To eliminate copy statements introduced by the previous algorithm we first need to solve a data flow analysis problem.

Let $cin[B]$ be the set of copy statements that (1) dominate $B$, and (2) their rhs’s are not rewritten before $B$.

$out[B]$ is the same, but with respect to the end of $B$.

$cgen[B]$ is the set of copy statements whose rhs’s are not rewritten before the end of $B$.

$ckill[B]$ is the set of copy statements not in $B$ whose rhs or lhs are rewritten in $B$.

We have the following equation:

$$cin[B] = \bigcap_{C \in PRED[B]} cgen[C] \cup (cin[C] - kcall[C])$$

Here we assume that $cin[S] = \emptyset$

Using the solution to this system of equations we can do copy propagation as follows:
Algorithm CP: Copy Propagation

Input: A flow Graph with use-definition chains, and cin[B] computed as just discussed above.

Output: A revised Flow graph

Method:

begin
  for each copy statement s: x=y do
    if for every use of x, s is in cin[B] and neither x nor y are redefined within B before the use then
      remove s and replace all uses of x by y.
    fi
  od
end
**Detection of Loop-Invariant Computation**

*Input.* A loop L. Use-definition chains

*Output.* The set of statements in L that compute the same value every time they are executed

*Method*

```plaintext
begin
    Mark “invariant” those statements whose operands are all either constant or have their reaching definitions outside L
    while at least one statement is marked invariant do
        mark invariant those statements with the following property: Each operand is either (1) constant, or (2) have all their reaching definitions outside L, or (3) have exactly one reaching definition, and that definition is a statement in L marked invariant.
    od
end
```

Once the “invariant” marking is completed statements can be moved out of the loop. The statements should be moved in the order in which they were marked by the previous algorithm. These statements should satisfy the following conditions:

1. All their definition statements that were within L should have been moved outside L.
2. The statement should dominate all exits of L
3. The lhs of the statement is not defined elsewhere in L, and
4. All uses in L of the lhs of the statement can only be reached by the statement.