Mathematical Programming manuscript No.
(will be inserted by the editor)

Uwe Naumann

On Optimal Jacobian Accumulation for Single Expression Use Programs

the date of receipt and acceptance should be inserted later

Abstract. ADIFOR and ADIC, the widely used software tools for Automatic Differentiation, use assignment-level reverse mode to compute local gradients of scalar assignment. This pre-accumulation often results in very efficient forward-mode derivative code. Scalar assignments belong to the class of Single Expression Use (SEU) programs. There, the values of all intermediate variables are read exactly once. Based on several theoretical results, we derive an algorithm for generating optimal Jacobian code for SEU programs. The number of floating-point operations performed during the accumulation of the Jacobian is minimized.

Key words. Single Expression Use Programs, Accumulation of Jacobian Matrices, Minimal Number of Arithmetic Operations

1. Introduction

We use a scalar assignment to introduce the theory required to prove the results of this paper. Consider `y=exp(x(2)*x(1)*sin(x(1)))`, where `y` is a scalar and `x` is a vector in $\mathbb{R}^n$ with $n = 2$. Its structure can be visualized as a directed acyclic graph (dag) with integer vertices, as shown on the left side of Figure 1. This corresponds to a decomposition of the statement into a so-called code list

```
v(3)=sin(x(1));
v(4)=x(1)*v(3);
v(5)=x(2)*v(4);
v(6)=exp(v(5))
```

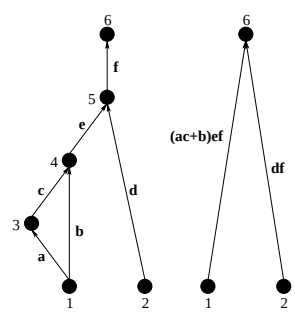

Fig. 1. Linearized dag

by assigning the results of all elemental functions φ_j, $j = n+1, \ldots, q$, (here $j = 3, \ldots, 6$ and $\varphi_j \in \{\texttt{exp}, \texttt{sin}, *\}$) to unique intermediate variables (e.g., $\texttt{v(3)}, \ldots, \texttt{v(6)}$). Suppose one is interested in the gradient of `y` with respect to the input vector `x` evaluated at some point $\texttt{x} = \texttt{x}_0$. Tools for *Automatic Differentiation* (AD) [8] can generate derivative code for computing this gradient by changing the semantics of the statement.

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Av., Argonne, IL 60439, e-mail: `naumann@mcs.anl.gov`

Forward Mode AD

```
v(3)=sin(x(1))
d_v(3)=d_x(1)*cos(x(1))
v(4)=x(2)*x(1)
d_v(4)=d_x(2)*x(1)+x(2)*d_x(1)
v(5)=v(4)*v(3)
d_v(5)=d_v(4)*v(3)+v(4)*d_v(3)
y=exp(v(5))
d_y=y*d_v(5)
```

Reverse Mode AD

```
v(3)=sin(x(1))
v(4)=x(2)*x(1)
v(5)=v(4)*v(3)
y=exp(v(5))
a_v(5)=y*a_y
a_v(4)=x(2)*a_v(5)
a_v(3)=x(1)*a_v(4)
a_x(2)=v(4)*a_v(5)
a_x(1)=v(3)*a_v(4)+cos(x(1))*a_v(3)
```

Fig. 2. Derivative Codes

Instead of computing the function value $\mathtt{y} = f(\mathtt{x})$ the transformed statement evaluates the inner product $(f', \mathtt{d\_x})$ in *forward mode* or the product $\mathtt{a\_y} \cdot f'$ when *reverse mode* is used [8, Chapter 3]. The vector $f' \in I\!R^n$ denotes the gradient of $\mathtt{y}$ with respect to $\mathtt{x}$. Furthermore, $\mathtt{d\_x} \in I\!R^n$ and $\mathtt{a\_y} \in I\!R$. The derivative codes resulting from applying AD to the example assignment are displayed in Figure 2. The values

$$\mathtt{d\_v(j)} = \sum_{\mathtt{i:\ v(i)\ \text{is argument of}}\ \varphi_j} \frac{\partial \varphi_j}{\partial \mathtt{v(i)}} \cdot \mathtt{d\_v(i)}$$

and

$$\mathtt{a\_v(k)} = \sum_{\mathtt{i:\ v(k)\ \text{is argument of}}\ \varphi_i} \frac{\partial \varphi_i}{\partial \mathtt{v(k)}} \cdot \mathtt{a\_v(i)}$$

are propagated forward for $\mathtt{j} = n+1, \ldots, q$ and backward for $\mathtt{k} = q-1, \ldots, 1$, respectively. Following the standard notation as in [8], we write $\mathtt{i} \prec \mathtt{j}$ if $\mathtt{v(i)}$ is an argument of $\varphi_{\mathtt{j}}$. Furthermore, we define $P_{\mathtt{j}} = \{\mathtt{i} : \mathtt{i} \prec \mathtt{j}\}$ and $S_{\mathtt{j}} = \{\mathtt{k} : \mathtt{j} \prec \mathtt{k}\}$. The gradient f' can be obtained either by executing the code generated by forward mode AD twice for $\mathtt{d\_x}$ equal to $(0,1)$ and $(1,0)$ or by simply setting $\mathtt{a\_y}$ equal to 1 in the *adjoint* code generated using the reverse mode of AD. In general, the complexity of computing the whole gradient in forward mode is $O(n)$, whereas the same task can be completed in reverse mode at a small constant multiple of the cost for evaluating the function itself [8, Rule 4]. This result is exploited by the AD tools ADIFOR 2.0 [2] and ADIC 1.1 [10]. Both tools use the forward mode to transform programs for evaluating a vector function $F : I\!R^n \to I\!R^m$ into derivative code for computing products $\dot{Y} = F' \cdot \dot{X} \in I\!R^{m \times l}$ of the Jacobian F' and a so-called *seed matrix* $\dot{X} \in I\!R^{n \times l}$. At the statement level these tools can perform a reverse mode accumulation of the local gradient, thus saving a significant number of floating-point operations (flops). For example, when accumulating F' itself without exploiting a possible structural sparsity (see [7], [16], and [4] for other approaches) $\dot{X}$ must be equal to the identity in $I\!R^n$. In forward mode the number of flops performed for our example assignment (which is assumed to be part of the evaluation program for F) would be $7n$ ($6n$ multiplications and n additions), where the number of edges in the dag is equal to 6. Assignment-level reverse-mode preaccumulation of the local gradient would

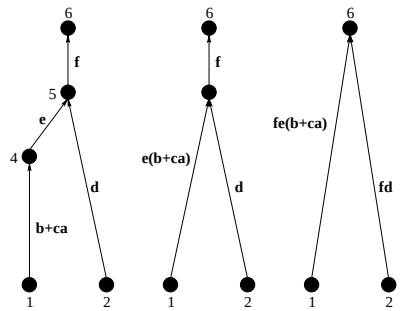

Fig. 3. Vertex Elimination Sequence (3,4,5)

take 5 multiplications and 1 addition followed by a vector-matrix product that would cost $2n$ multiplications and n additions. Here we assume that the trivial multiplication of `y` with one is not performed in the code on the right of Figure 2, which yields `a_v(5)=y`. In many real-world applications n tends to become very large [11, 17, 20], which makes the improvements in the number of flops approach a factor of 3.

Automatic Differentiation is based on the idea that the numerical values of the local partial derivatives

$$c_{j,i} \equiv \frac{\partial \varphi_j}{\partial \mathtt{v(i)}}(\mathtt{v(i)}) \in I\!R \tag{1}$$

of all elementary functions with respect to its arguments can be computed at the current point $\mathbf{x}_0$ and that they can be attached to the corresponding edges in the dag. In our example $\mathbf{a} = c_{3,1} = \cos(\mathtt{v(1)})$, $\mathbf{b} = c_{4,1} = \sin(\mathtt{v(1)})$, $\mathbf{c} = c_{4,3} = \mathtt{v(1)}$, $\mathbf{d} = c_{5,2} = \mathtt{v(4)}$, $\mathbf{e} = c_{5,4} = \mathtt{v(2)}$, $\mathbf{f} = c_{5,4} = \exp(\mathtt{v(5)}) = \mathtt{v(6)}$. The gradient can be computed by using either forward or reverse mode, as shown in Figure 2 or by applying elimination techniques [14] to this *linearized* version of the dag. For example, the elimination of all intermediate vertices (3,4, and 5) transforms the dag into the directed complete bipartite graph $K_{n,1}$ shown on the right side of Figure 1. The graphs resulting from the elimination of vertex 4 followed by 5 and 3 are displayed in Figure 4. When eliminating j, new edges are introduced connecting the predecessors of j with its successors. A new edge (i,k) is labeled with the product of the labels of (j,k) and (i,j). If either $c_{k,j}$ or $c_{j,i}$ or both are trivial, that is equal to 1 or -1, then the multiplication is not performed explicitly. Parallel edges are *merged*, and the corresponding edges labels are added. For example, $c_{6,1} = c_{6,1} + c_{6,3} \cdot c_{3,1}$ when eliminating 3 in the second graph in Figure 4. Finally, j is removed together with its incident edges. The number of scalar multiplications involved in the elimination of j is referred to as the *Markowitz degree* of j, and it is equal to $|P_j||S_j|$. Here, $|.|$ denotes the cardinality of a set. The number of additions is equal to the number of edges $(i,k) \in E$ that existed in $\mathbf{G}$ before the elimination of j and for which $i \prec j$ and $j \prec k$.

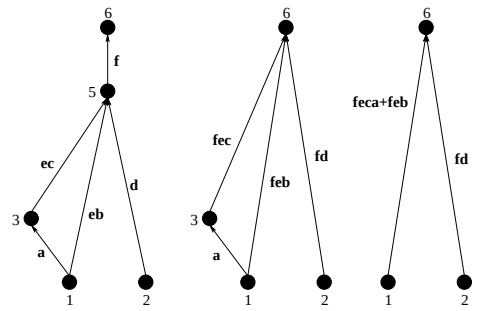

Fig. 4. Vertex Elimination Sequence (4,5,3)

The elimination of a vertex j is equivalent to the simultaneous *front elimination* of all edges leading into it. Similarly, the elimination of j is equivalent to the simultaneous *back elimination* of all edges emanating from it. An edge (i,j) is front eliminated by connecting i with all successors of j. For all successors k of j, the new edges (i,k) are labeled with $c_{k,i} = c_{k,j}c_{j,i}$. If (i,k) existed before, then $c_{k,i} = c_{k,i} + c_{k,j}c_{j,i}$. The edge (i,j) is removed after this. The number of multiplications required to front eliminate (i,j) is equal to $|S_j|$. An addition is performed for all (i,k) that existed in $\mathbf{G}$ before the elimination of (i,j) such that $j \prec k$. Analogously, an edge (j,k) is back eliminated by connecting all predecessors of j with k. For all predecessors i of j the new edge (i,k) is labeled with $c_{k,i} = c_{k,j}c_{j,i}$. Again, the label becomes $c_{k,i} = c_{k,i} + c_{k,j}c_{j,i}$ if (i,k) existed before. Finally, (j,k) is removed. The number of multiplications required to back eliminate (j,k) is equal to $|P_j|$. Additions are performed for all (i,k) that existed in $\mathbf{G}$ before the elimination of (j,k) such that $i \prec j$. Newly generated edges are referred to as fill-in.

Figure 5 shows the dags resulting from the application of the edge elimination sequence $[(1,4)$ front, $(1,3)$ front, $(5,6)$ back$]$ to the dag of the example assignment. Notice that in dags of SEU programs the back elimination of an edge is always equivalent to the elimination of its source. This follows immediately from all intermediate vertices having exactly one successor. Elimination techniques in linearized computational graphs are discussed in detail in [15]. There we prove both the structural and numerical correctness of the corresponding graph transformations.

The computation of Jacobian matrices by vertex elimination was first proposed in [9]. Regardless of the order in which the intermediate vertices are eliminated (there are $3! = 6$ different ways in our example), one gets the same value for the gradient of $\mathbf{y}$ with respect to $\mathbf{x}$.[1] However, the number of flops performed varies. For example, the elimination sequence illustrated in Figure 4 involves six multiplications and one addition. Eliminating 3 followed by 5 and 4 decreases

[1] This follows immediately from the associativity of the chain rule in $\mathbb{R}$. Our discussion is based on the assumption that the same is true for floating-point numbers. To our knowledge, the actual implications for the stability of numerical algorithms have not been investigated so far.

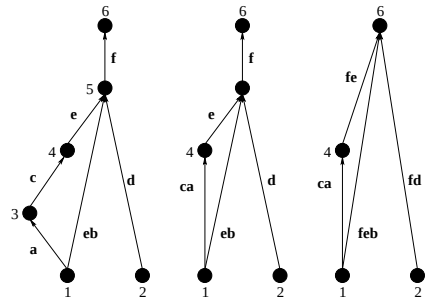

Fig. 5. Edge Elimination Sequence [(1, 4) front, (1, 3) front, (5, 6) back]

the number of multiplication by two. This process can be followed in Figure 3. In fact, it represents the gradient computation that minimizes the number of flops performed.

The remainder of this paper is structured as follows. Section 2 formalizes the framework established in Section 1 and it gives a formulation of the combinatorial optimization problem that is solved in Section 3 which represents the heart of the paper. The proofs of several theoretical results lead to a deterministic polynomial algorithm for solving the problem for SEU programs. A very efficient version of it is derived for a significant special case. The paper concludes with a case study in Section 4.

2. Formal Framework

We consider programs representing nonlinear vector functions mapping n *independent* onto m *dependent* variables as $\mathbf{y} = F(\mathbf{x})$, where $\mathbf{x} \equiv (x_1, \ldots, x_n) \in I\!R^n$ and $\mathbf{y} = (y_1, \ldots, y_m) \in I\!R^m$. The objective is to transform the program into a set of statements for computing the Jacobian matrix

$$F' = F'(\mathbf{x}_0) = \left(\frac{\partial y_i}{\partial x_j}(\mathbf{x}_0) \right)_{i=1,\ldots,m,\ j=1,\ldots,n}$$

of F with respect to the n inputs for a given argument $\mathbf{x}_0$ and using a minimal number of flops. This problem is referred to as the Optimal Jacobian Accumulation (OJA) problem[2] in [15].

As pointed out in Section 1, we assume that F can be decomposed into a sequence of assignments of the values of scalar elemental functions φ_j to unique intermediate variables v_j, $j = n+1, \ldots, n+p$, or the dependent variables $y_i \equiv v_i$, where $i = n+p+1, \ldots, n+q$ and $q = p+m$. The resulting code list is given as

$$(I\!R \ni)\ v_j = \varphi_j(v_i)_{i \prec j} \quad , \tag{2}$$

[2] In [15] we consider the number of fused multiply-add operations that is equivalent to the number of multiplications. In the paper at hand, this restriction can be relaxed, as optimal values for both the number of additions and the number of multiplications can be obtained for SEU programs.

where $j = n+1, \ldots, q$. The transitive closure of the precedence relation $\prec$ is denoted by $\prec^+$. All variables in F are numbered consistently according to $\mathcal{I} : V \to \{1, \ldots, q\}$ where $i \prec^+ j \Rightarrow \mathcal{I}(v_i) < \mathcal{I}(v_j)$.

Since the differentiation of F is based on the differentiability of its elemental functions, it will be assumed that the φ_j, $j = n+1, \ldots, q$, have jointly continuous partial derivatives

$$c_{j,i} \equiv \frac{\partial}{\partial v_i} \varphi_j(v_k)_{k \prec j} \quad , \quad i \prec j \quad , \tag{3}$$

on open neighborhoods $\mathcal{D}_j \subset I\!R^{n_j}$, $n_j \equiv |\{i | i \prec j\}|$, of their domain. In this case the numerical values of all $c_{j,i}$ can be computed in parallel with the function value $F(\mathbf{x}_0)$ at the current argument by a single evaluation of F. W.l.o.g., we assume that the $c_{j,i}$ are non-trivial. A dag $\mathbf{G} = (V, E)$ with $V = \{i : v_i \in F\}$ and $(i, j) \in E$ if $i \prec j$ can be associated with F. Its vertex set $V = X \cup Z \cup Y$ is such that $X = \{1, \ldots, n\}$, $Z = \{n+1, \ldots, n+p\}$, and $Y = \{n+p+1, \ldots, n+q\}$. We assume $\mathbf{G}$ to be linearized in the sense that the $c_{j,i}$ are attached to their corresponding edges, that is, (i, j) is labeled with $c_{j,i}$. The linearized dag is also referred to as the *c-graph* [14].

In c-graphs of programs whose intermediate values are read exactly once, all intermediate vertices have exactly one successor. This property is crucial for the derivation of the main results in this paper. Often the elemental functions are at most binary, which implies a stronger result and a more efficient algorithm.

Methods for accumulating Jacobian matrices of general vector functions by applying elimination methods to c-graphs are discussed in [15]. *Vertex* [9], *edge* [13], [8], and *face* [15] elimination techniques are used to minimize the number of flops involved in the computation of the Jacobian. This represents a complex combinatorial optimization problem [14] that is conjectured to be NP complete [9], [3]. We show in Section 3 that in the context of SEU programs it is sufficient to consider vertex elimination as described in Section 1. The Jacobian can be computed by eliminating the p intermediate vertices in the corresponding c-graph $\mathbf{G}$. There are $p!$ different vertex elimination orderings and possibly several out of them minimize the overall flops count. The labels on the edges of the resulting directed complete bipartite graph $K_{n,m}$ represent the entries of the Jacobian matrix as shown in [9].

Memory access patterns may play an important role in minimizing the overall run time of a derivative code. The consideration of all factors influencing the generation of efficient derivative code results in an even more complicated combinatorial optimization problem. Its solution depends strongly on the computational platform [18] and offers a variety of interesting research topics. We consider the investigation of the OJA problem as a very important part of the research in efficient derivative code generation.

3. Optimal Jacobian Accumulation for SEU Codes

To derive an algorithm for accumulating Jacobians of SEU codes efficiently, we require additional terminology. For $A, B \subset V$, $A \cap B = \varnothing$, an *A-B-separating (vertex) set* $S_{A,B}$ is a subset of V for which $S_{A,B} \cap A = \varnothing$ and whose removal separates A from B, that is, in the graph induced by $V \setminus S_{A,B}$ there is no path connecting any vertex in B with a vertex in A. For example, $\{4,5\}$ is a $\{6\}$-$\{1,2\}$-separating set in the example dag in Figure 1. For a given intermediate vertex $j \in \{n+1, \ldots, n+p\}$ we denote a minimal $\{j\}$-$\{1, \ldots, n\}$-separating set by $\underline{P_j}$. Similarly, we denote a minimal $\{n+p+1, \ldots, n+q\}$-$\{j\}$-separating set by $\underline{S_j}$. Obviously, the size of any minimal $\{n+p+1, \ldots, n+q\}$-$\{j\}$-separating set is equal to one for SEU codes and for all $j \in \{n+1, \ldots, n+p\}$. For example, $\{4\}$ is a minimal $\{6\}$-$\{3\}$-separating set in Figure 1. The same is true for $\{5\}$.

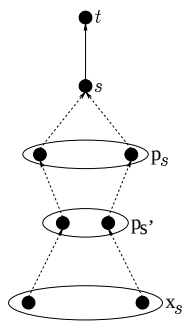

Fig. 6. Lemma 1

The following result states a lower bound for the number of occurrences of some local partial derivative as a factor in the final derivative code. This result applies to the computation of Jacobian matrices in general. Later we show that this lower bound can actually be reached for SEU codes. Thus, we can derive an algorithm for solving the OJA problem for SEU codes deterministically.

Lemma 1. *For $s \prec t$ the value $\min(|\underline{P_s}|, |\underline{S_t}|)$ is a lower bound for the number of scalar floating point multiplications involving the local partial derivative $c_{t,s}$ as a factor.*

Proof. By the chain rule

$$g_t \equiv \frac{\partial v_t}{\partial \mathbf{x}_t} = \frac{\partial v_t}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{x}_t} \quad , \tag{4}$$

where $\mathbf{p}_t \in I\!R^{|P_t|}$ is the vector of all predecessors of t in $\mathbf{G}$. In other words, the row vector

$$\frac{\partial v_t}{\partial \mathbf{p}_t} \equiv (c_{t,p_1}, \ldots, c_{t,p_{|P_t|}}) \in I\!R^{|P_t|}$$

contains the local partial derivatives of v_t with respect to its predecessors. These values are attached as labels to all edges leading into t in $\mathbf{G}$. The components of $\mathbf{x}_t$ are all those v_i for which $i \in X$ and $i \prec^+ t$. Focusing on the occurrences of $c_{t,s}$ in g_t, consider

$$g_{t,s} \equiv \frac{\partial v_t}{\partial v_s} \cdot \frac{\partial v_s}{\partial \mathbf{x}_s} = c_{t,s} \cdot \frac{\partial v_s}{\partial \mathbf{x}_s} \quad . \tag{5}$$

Let P'_s be an s-$\mathbf{x}_s$-separating set in $\mathbf{G}$. With $\mathbf{p}'_s \in I\!R^{|P_s|}$ representing the vector of the corresponding intermediate variables, Equation (5) can be written as

$$g_{t,s} = c_{t,s} \cdot \frac{\partial v_s}{\partial \mathbf{p}_s} \cdot \frac{\partial \mathbf{p}_s}{\partial \mathbf{p}'_s} \cdot \frac{\partial \mathbf{p}'_s}{\partial \mathbf{x}_s} \quad . \tag{6}$$

This is an immediate consequence of the chain rule. A graphical illustration is given in Figure 6. Equation (6) can be evaluated in various ways because of the associativity of matrix multiplication. We are interested in an order that minimizes the size of the vector that is multiplied by $c_{t,s}$. Consider

$$g_{t,s} = \left(c_{t,s} \cdot \left(\frac{\partial v_s}{\partial \mathbf{p}_s} \cdot \frac{\partial \mathbf{p}_s}{\partial \mathbf{p}'_s} \right) \right) \cdot \frac{\partial \mathbf{p}'_s}{\partial \mathbf{x}_s} \quad . \tag{7}$$

The innermost factors are the local gradient

$$\frac{\partial v_s}{\partial \mathbf{p}_s} \in I\!R^{|P_s|}$$

and the local Jacobian

$$\frac{\partial \mathbf{p}_s}{\partial \mathbf{p}'_s} \in I\!R^{|P_s| \times |P'_s|} \quad .$$

Their product yields

$$\frac{\partial v_s}{\partial \mathbf{p}'_s} = \frac{\partial v_s}{\partial \mathbf{p}_s} \cdot \frac{\partial \mathbf{p}_s}{\partial \mathbf{p}'_s} \in I\!R^{|P'_s|} \quad ,$$

which is then multiplied by the scalar $c_{t,s}$. Consequently, the number of occurrences of $c_{t,s}$ as a factor in one of the scalar products is minimized if and only if $|P'_s|$ is minimal, that is, if P'_s is a minimal s-$\mathbf{x}_s$-separating set in $\mathbf{G}$.

It remains to note that the computation of

$$\frac{\partial v_t}{\partial \mathbf{p}'_s} = c_{t,s} \cdot \frac{\partial v_s}{\partial \mathbf{p}'_s}$$

is equivalent to the back elimination of the edge (s,t) in $\mathbf{G}$ as described in [15]. This implies that once this operation has been performed, the factor $c_{t,s}$ will not appear in any other product unless it gets regenerated as fill-in. However, since we are proving a lower bound for the cost of eliminating an edge (s,t), it is save to assume that (s,t) is in fact the only path connecting s and t in $\mathbf{G}$.

A similar argument can be applied to

$$\bar{g}_{t,s} \equiv \frac{\partial \mathbf{y}_t}{\partial v_s} = \frac{\partial \mathbf{y}_t}{\partial \mathbf{s}'_t} \cdot \left(\left(\frac{\partial \mathbf{s}'_t}{\partial \mathbf{s}_t} \cdot \frac{\partial \mathbf{s}_t}{\partial v_t} \right) \cdot c_{t,s} \right) \quad . \tag{8}$$

One observes that

$$\frac{\partial \mathbf{s}'_t}{\partial \mathbf{s}_t} \cdot \frac{\partial \mathbf{s}_t}{\partial v_t} \in I\!R^{|S'_t|} \quad ,$$

making $|S'_t|$ equal to the number of occurrences of $c_{t,s}$ as factor in the computation of $\bar{g}_{t,s}$. This number is minimized if and only if S'_t is a minimal t-$\mathbf{y}_t$-separating set in $\mathbf{G}$. The edge (s,t) is front eliminated as a result of this computation.

Multiplications involving $c_{t,s}$ will appear as long as $(s,t) \in E$ has not been either front or back eliminated. Earlier we showed that the minimal cost of doing this is $\min(|\underline{P_s}|, |\underline{S_t}|)$, which completes the proof. $\square$

A polynomial algorithm for computing minimal separating sets is discussed in [19].

Lemma 2. *Given a c-graph* $\mathbf{G}$, *the value* $\sum_{j \in Z} |\underline{P_j}| \cdot |\underline{S_j}|$ *is a lower bound for the solution of the* OJA *problem.*

Proof. Consider an intermediate vertex $j \in Z$. A consequence of Lemma 1 is that at least $|\underline{P_j}|$ in-edges of j must be front eliminated at a minimal cost of $|\underline{S_j}|$ multiplications each or $|\underline{S_j}|$ out-edges are back eliminated at a minimal cost of $|\underline{P_j}|$ multiplications. In any case, $|\underline{P_j}||\underline{S_j}|$ is a lower bound for the number of multiplications required to eliminate j. This applies to all intermediate vertices in $\mathbf{G}$. □

A c-graph is called *absorption free* if any two vertices are connected by exactly one path.

Lemma 3. *Reverse vertex elimination solves the OJA problem for SEU programs whose c-graphs are absorption free.*

Proof. Notice that such c-graphs are sets of trees over a common set of leafs $\{1, \ldots, n\}$ and disjoint vertex and edge sets otherwise. In any tree, $|\underline{S_j}| = 1$ for $j = 1, \ldots, n+p$. Obviously, $|P_j|$ is equal to $|\underline{P_j}|$ for all $j = n+1, \ldots, q$. Moreover, eliminating all intermediate vertices in reverse order ensures that both the numbers of predecessors and successors of all vertices remain constant. Consequently, the number of multiplications involved in the elimination of all intermediate vertices is equal to the lower bound established in Lemma 2. The number of additions is equal to zero. □

Theorem 1. *Let* $|\{[i \to j]\}|$ *denote the number of distinct paths connecting two vertices* $i \in X$ *and* $j \in Y$. *The number of additions performed by any elimination sequence when applied to the c-graph of an SEU program is equal to* $\alpha = \sum_{i \in X, j \in Y} (|\{[i \to j]\}| - 1)$.

Proof. According to a result in [12], an entry of the Jacobian of F can be accumulated as

$$F'(i,j) = \sum_{[i \to j]} \prod_{(k,l) \in [i \to j]} c_{l,k} \quad , \tag{9}$$

where $i \in X$ and $j \in Y$. For SEU graphs this number can neither be increased nor decreased. The former is obvious, since Equation (9) represents the sum over all distinct paths. The number of additions could be increased only if there were more distinct paths.

Suppose that some elimination sequence could decrease the number of additions. This is the case only if some sum $a + b$ appears as a common subterm in two or more Jacobian entries, for example in $c(a+b)$ and $d(a+b)$. In that case, the substitution of $e = a + b$ in both expression decreases the number of additions by one. Obviously, a sum cannot appear twice in the same Jacobian entry. W.l.o.g., let $c_{k,i} + c_{k,j}c_{j,i}$ be a common subexpression in two or more Jacobian

entries. First, we must have $i \in X$ since only independent vertices are allowed to have more than one successor. In that case, however, k or some k' with $k \prec^+ k'$ would need to have at least two successors for $c_{k,i} + c_{k,j}c_{j,i}$ to appear in two different Jacobian entries. This yields a contradiction to the definition of SEU programs.

Consequently, $|\{[i \to j]\}| - 1$ additions must be performed for each pair $i \in X$ and $j \in Y$. □

An immediate consequence of Theorem 1 is that the number of additions that must be performed to compute the gradient of a scalar assignment is equal to $\sum_{i \in X} (|S_i| - 1)$.

Theorem 2. *Given a linearized dag $\mathbf{G} = (V, E)$ of an SEU program implementing a vector function $\mathbf{y} = F(\mathbf{x})$, the Jacobian*

$$F'(\mathbf{x}) = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

can be computed with a minimal number of flops as follows:

1. *Do for $j = n+1, \ldots, n+p$.*
   (a) *Compute minimal $\{1, \ldots, n\}$-$\{j\}$-separating set $\underline{P_j}$.*
   (b) *If $|\underline{P_j}| < |P_j|$, then compute the local gradient $\partial v_j / \partial \mathbf{x}$ by reverse vertex elimination.*
2. *Compute $\partial \mathbf{y} / \partial \mathbf{x}$ by reverse vertex elimination.*

Proof. With Theorem 1 and Lemma 2, $\alpha + \sum_{j=n+1,\ldots,n+p} |\underline{P_j}|$ is a lower bound for the solution of the OJA problem for SEU programs, since $|\underline{S_j}| = 1$ for $j = n+1, \ldots, n+p$. Let j be the first intermediate vertex such that $|\underline{P_j}| < |P_j|$. The computation of $\partial v_j / \partial \underline{P_j}$ by vertex elimination would decrease the in-degree of j by $|P_j| - |\underline{P_j}|$. Since $|\underline{P_i}| = |P_i|$ and $|S_i| = 1$ for $i = n+1, \ldots, j-1$, the local gradient $\partial v_j / \partial \underline{P_j}$ can be computed at a minimal cost by reverse vertex elimination. Application of this argument to all $j = n+1, \ldots, p+n$ followed by a global reverse vertex elimination ensures that all intermediate vertices are eliminated at minimal cost. Consequently, the algorithm results in a vertex elimination sequence that reaches the lower bound established by Lemma 2. Both the number of additions and the number of multiplications become minimal, and therefore the algorithm solves the OJA problem for SEU programs. □

Often the elemental functions used are at most binary. In this case the Jacobian accumulation algorithm can be simplified to become considerably more efficient.

Theorem 3. *Given a linearized dag $\mathbf{G} = (V, E)$ of an SEU program implementing a vector function $\mathbf{y} = F(\mathbf{x})$ with $|P_j| \leq 2$ for $j \in V$. Then the Jacobian $F'(\mathbf{x})$ can be computed by using a minimal number of flops as follows:*

1. *Eliminate all vertices with Markowitz degree equal to one in forward mode.*
2. *Compute $\partial \mathbf{y} / \partial \mathbf{x}$ by using reverse vertex elimination.*

Proof. Again, the number of additions performed is equal to α. Moreover, the computation of a minimal $\{1,\ldots,n\}$-$\{j\}$-separating set becomes very simple. The only situation in which $|\underline{P_j}| < |P_j|$ could possibly occur is $|P_j| = 2$ and $|\underline{P_j}| = 1$. This implies that $\underline{P_j} = \{i\}$ for some $i \in \{1,\ldots,n\}$ and that $\{i\}$ is also a minimal $\{1,\ldots,n\}$-$\{k\}$-separating set for all k with $i \prec^+ k$ and $k \prec^+ j$. When considering j all these k must have a Markowitz degree of one and can therefore be eliminated at minimal cost. The in-degree of j thus itself becomes equal to one. Finally, once all vertices j with $|P_j| = 1$ have been eliminated, the remaining remaining vertices can be eliminated at minimal cost in reverse order. □

From the AD literature [1], [5], [6], [8], we know that the complexity of derivative codes for computing gradients of scalar functions $f : I\!R^n \to I\!R$ based on forward (vertex elimination) mode is $O(n)$. According to the *cheap gradient principle* [8, Rule 4], the gradient of f can be computed at a small constant multiple of the cost of evaluating f itself. This is supported by the numerical results presented in Section 4, and it is exploited in the statement-level reverse mode featured by the AD tools ADIFOR and ADIC. The main motivation for the research that led to the results in this paper is the fact that reverse mode is not optimal for computing gradients in general. In particular, there is the following result.

Theorem 4. *An optimal derivative code undercuts the number of flops performed by a reverse vertex elimination sequence by less than a factor of two for SEU programs.*

Proof. Notice first that for SEU programs the number of additions performed is the same for both a reverse and an optimal vertex elimination sequence. By Lemma 2 in an optimal derivative code for an SEU program, each intermediate vertex $j \in \{n+1,\ldots,n+p\}$ in the corresponding c-graph is eliminated at the cost of $|\underline{P_j}|$ scalar multiplications. In a reverse vertex elimination sequence the elimination of the same vertex involves $|P_j|$ multiplications. In SEU programs the elimination of some intermediate vertex can lead only to the indegree of its single successor being decreased by at most one. Notice that this costs at least one multiplication. Consider the "worst case" where the elimination of every vertex leads to a decrease of the in-degree of its successor. This is the case only for graphs shaped as shown on the left of Figure 7. For p intermediate variables the factor between reverse and optimal vertex elimination stays below two as p approaches ∞. □

4. Case Study and Conclusions

Consider the following two assignments,

$$\mathtt{y} = \mathtt{x}(1) * \mathtt{x}(2) * \ldots * \mathtt{x}(\mathtt{n}) \tag{10}$$

and

$$\mathtt{y} = \mathtt{sin}(\mathtt{x}) * \mathtt{x} * \ldots * \mathtt{x} \quad , \tag{11}$$

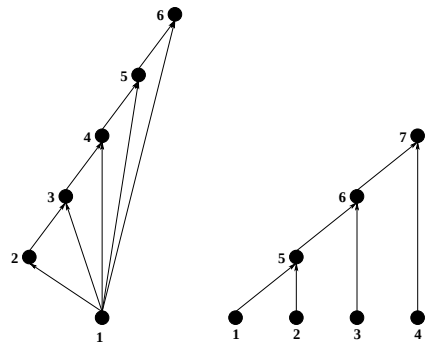

Fig. 7. Case Study

where `sin(x)` is multiplied `n` times by `x`. The dags are shown in Figure 7 for `n` equal to 4. Our numerical tests are based on the 10^6 executions of the assignments and the corresponding derivative codes for `n=20` on an INTEL Pentium III processor.

For statement (10) we compared forward with reverse vertex elimination and got the following results:

	Flops	Runtime (sec)
Function	20	22
Forward	209	293
Reverse	36	55

We have also listed the runtime for executing the original assignment 10^6 times. The execution of the reverse vertex elimination derivative code takes 2.5 times longer than the original code. The factor of 5.8 between forward and reverse suggested by considering the number of flops can nearly be carried over to the runtime, which exhibits a corresponding factor of 5.3. Reverse mode is optimal in this case by Lemma 3.

For the second example, statement (11), we have compared the results for the reverse vertex elimination derivative code with the optimal vertex elimination derivative code (which happens to be equivalent to forward vertex elimination in this particular case).

	Flops	Runtime (sec)
Function	21	43
Optimal	40	51
Reverse	59	66

The operations count suggests a factor of almost 1.5 between optimal and reverse which cannot be observed when considering the runtime. The reason may be the fact that the reverse vertex elimination sequence requires only one additional temporary variable. It is well known that memory accesses are, in general, more expensive than scalar flops. Therefore they are likely to dominate the runtime in our examples.

We conclude that investigations into optimizing derivative code from the viewpoint of memory accesses are very important. In connection with the algorithms presented in this paper this research is likely to lead to a significant speedup of the derivative codes generated by next-generation AD software tools.

Acknowledgment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.
2. C. Bischof, A. Carle, P. Khademi, and A. Maurer. The ADIFOR 2.0 system for Automatic Differentiation of Fortran 77 programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.
3. C. Bischof and M. Haghighat. Hierarchical approaches to Automatic Differentiation. In *[1]*, pages 82–94, 1996.
4. T. Coleman and A. Verma. Structure and efficient Jacobian calculation. In *[1]*, pages 149–159. SIAM, 1996.
5. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
6. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.
7. A. Curtis, M. Powell, and J. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.
8. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
9. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[6]*, pages 126–135, 1991.
10. P. Hovland and B. Norris. Users' guide to ADIC 1.1. Technical Memorandum ANL/MCS-TM-225, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.
11. J. Marshall, C. Hill, L. Perelman, and A. Adcroft. Hydrostatic, quasi-hydrostatic and nonhydrostatic ocean modeling. *J. Geophysical Research*, 102, C3:5,733–5,752, 1997.
12. W. Miller and C. Wrathall. *Software for Roundoff Analysis of Matrix Algorithms.* Academic Press, New York, 1980.
13. U. Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs.* PhD thesis, Technical University Dresden, Feb. 1999.
14. U. Naumann. Elimination techniques for cheap Jacobians. In *[5]*, pages 247–253, 2002.
15. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming*, 2002. Under review.
16. G. Newsam and J. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Dis. Meth.*, 4:404–417, 1983.
17. S. Park, K. Droegemeier, and C. Bischof. Automatic differentiation as a tool for sensitivity analysis of a convective storm in a 3-D cloud model. In *[1]*, pages 205–214. SIAM, 1996.
18. M. Tadjouddine, S. Forth, J. Pryce, and J. Reid. Performance issues for vertex elimination methods in computing Jacobians using Automatic Differentiation. In *Proceedings of the ICCS 2000 Conference*, volume 2330 of *Springer LNCS*, pages 1077–1086, 2002.
19. Jin Tian, Azaria Paz, and Judea Pearl. Finding minimal D-separators. Technical Report 980007, University of California, Los Angeles, 1998.
20. S. Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach.* LNCSE. Springer, New York, 1999.