Electronic Transactions on Numerical Analysis.
Volume 21, pp. 134-150, 2005.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

OPTIMALITY-PRESERVING ELIMINATION OF LINEARITIES IN JACOBIAN ACCUMULATION*

UWE NAUMANN† AND JEAN UTKE‡

Dedicated to Alan George on the occasion of his 60th birthday

Abstract. We consider a mathematical function that is implemented in a high-level programming language such as C or Fortran. This function is assumed to be differentiable in some neighborhood of a set of input arguments. For available local partial derivatives of the arithmetic operators and intrinsic functions provided by the programming language, the Jacobian of the function at the given arguments can be accumulated by using the chain rule. This technique is known as automatic differentiation of numerical programs.

Under the above assumptions the values of the local partial derivatives are well defined for given values of the inputs. A code for accumulating the Jacobian matrix that is based on the chain rule takes these partial derivatives as input and computes the nonzero entries of the Jacobian using only scalar multiplications and additions. The exploitation of the associativity of the chain rule or, equivalently, the algebraic properties of the corresponding field $(\mathbb{R}, *, +)$ – in particular, associativity of the multiplication and distributivity – to minimize the number of multiplications leads to a combinatorial optimization problem that is widely conjectured to be NP-hard. Several heuristics have been developed for its approximate solution. Their efficiency always depends on the total number of partial derivatives.

Linearities in the function lead to constant partial derivatives that do not depend on the input values. We present a specialized constant folding algorithm to decrease the size of the combinatorial problem in order to increase the efficiency of heuristics for its solution. Moreover, we show that this algorithm preserves optimality in the sense that an optimal solution for the reduced problem yields an objective value no worse than that of an optimal solution for the original problem.

Key words. Jacobian accumulation, linearities, constant folding

AMS subject classifications. 90C27, 26B10, 68N99

1. The Problem.

A given vector function

$$\boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$$

is implemented in a numerical program in some higher programming language such as C or Fortran. For the purpose of automatic differentiation (AD) [9], the numerical programs are represented by directed acyclic graphs (DAG) with elemental partial derivatives as edge labels. AD provides a variety of elimination techniques that allow propagation of derivative information. The choice of the technique is subject to the structure of $\boldsymbol{f}$ and application-dependent optimization criteria.

Linear parts of the function yield constant partial derivatives. Ignoring this fact leads to derivative code that performs operations at runtime that could potentially be done at compile-time. Relying on the constant-folding capabilities [13] of the compiler during the compilation of the derivative code may not be sufficient. Consider, for example, the statement

$$z = e^{c_1 \cdot (c_2 \cdot \sin(x) + c_3 \cdot \cos(y))} \quad .$$

Without taking into account that c_1, c_2, and c_3 are constants the elimination techniques discussed in Section 2 combined with the Lowest-Markowitz-Degree heuristic [11] generate the

*Received August 31, 2004. Accepted for publication October 25, 2005. Recommended by A. Pothen.

†Software and Tools for Computational Engineering, RWTH Aachen University, D-52056 Aachen, Germany (naumann@stce.rwth-aachen.de).

‡Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439–4844, USA (utke@mcs.anl.gov).

ETNA
Kent State University
etna@mcs.kent.edu

following code for the computation of the local gradient.

$$h = c_1 \cdot z; \quad \frac{\partial z}{\partial x} = h \cdot c_2 \cdot \cos(x); \quad \frac{\partial z}{\partial y} = -h \cdot c_3 \cdot \sin(y) \quad .$$

For the compiler of this code to recognize that $p_1 = c_1 \cdot c_2$ and $p_2 = c_1 \cdot c_3$ can be computed at compile-time followed by the generation of code for computing

$$\frac{\partial z}{\partial x} = z \cdot p_1 \cdot \cos(x); \quad \frac{\partial z}{\partial y} = -z \cdot p_2 \cdot \sin(y)$$

would require inlining of h in the original derivative code. To our knowledge general-purpose compilers do not perform this kind of extended static constant folding analysis. Exploitation of the domain-specific knowledge allows us to generate the optimal code directly.

The second and more significant motivation for the research that led to this paper is the complexity of static methods for computing efficient elimination sequences. Some of them are quadratic in the size of the directed acyclic computational graph of $\boldsymbol{f}$. Their efficiency (and often their effectiveness too) improves with a decreasing size of the graph.

Generally the code for $\boldsymbol{f}$ contains control flow (loops, branches) that does not allow the representation with a single DAG. Locally, for example within a basic block, a representative DAG can be constructed. For the purpose of this paper we assume a scenario where the accumulation of a local Jacobian is beneficial. In practical terms, we may encounter a frequently executed innermost loop body that has an evaluation cost that is relatively high compared to the cost of storing a local Jacobian. We find a highly efficient procedure to calculate the local Jacobians, which are then used in a fashion determined by the context.

The paper is structured as follows. Section 2 presents some essential background information. Section 3 represents the heart of the paper. We present new algorithms for the optimality-preserving elimination of linearities in Jacobian accumulation by constant folding. Numerical results are presented in Section 4 with the help of three case studies. Section 5 concludes the paper with a discussion of how the new algorithms are used in practice.

2. Background. In order to understand the main contribution of this paper we find it useful to present some background information on the context of preaccumulation of local Jacobians and the corresponding graph modification techniques. The main target application is the automatic generation of efficient tangent-linear and adjoint codes by automatic differentiation (AD). As in any static source-transformation approach the objective is to minimize the amount of computational work that has to be done at run-time by doing as much as possible at compile-time. For a formal introduction to the mathematical principles underlying AD, we refer the reader to [9, 20, 7, 4, 6]. A collection of links to ongoing research and tool development projects can be found on the AD community's website `www.autodiff.org`.

2.1. Preaccumulation of Local Jacobians. Conceptually, AD is based on a decomposition of the evaluation routine for $\boldsymbol{f}$ into a three-address code[1] of the form

$$v_k = \varphi_j(v_i, v_j) \tag{2.1}$$

for $k = 1, \ldots, p+m$ and $i, j = 1-n, \ldots, p$, $k > i, j$. The *elemental* functions φ_k, $k = 1, \ldots, p+m$, are assumed to have jointly continuous partial derivatives in a neighborhood of the current argument. The n *independent* variables $x_1, \ldots, x_n$ correspond to $v_{1-n}, \ldots, v_0 \in X$. We consider the computation of the first derivative of the *dependent* variables $y_1, \ldots, y_m$

[1]We assume that there are at most binary arithmetic operators and intrinsic functions. The generalization is trivial.

represented by m variables $v_{p+1}, \ldots, v_{p+m} \in Y$ with respect to the independents. The resulting $m \times n$ matrix is known as the *Jacobian matrix* of $\boldsymbol{f}$. For notational simplicity and without loss of generality we assume that the dependent variables are mutually independent. This situation can always be reached by introducing auxiliary assignments.

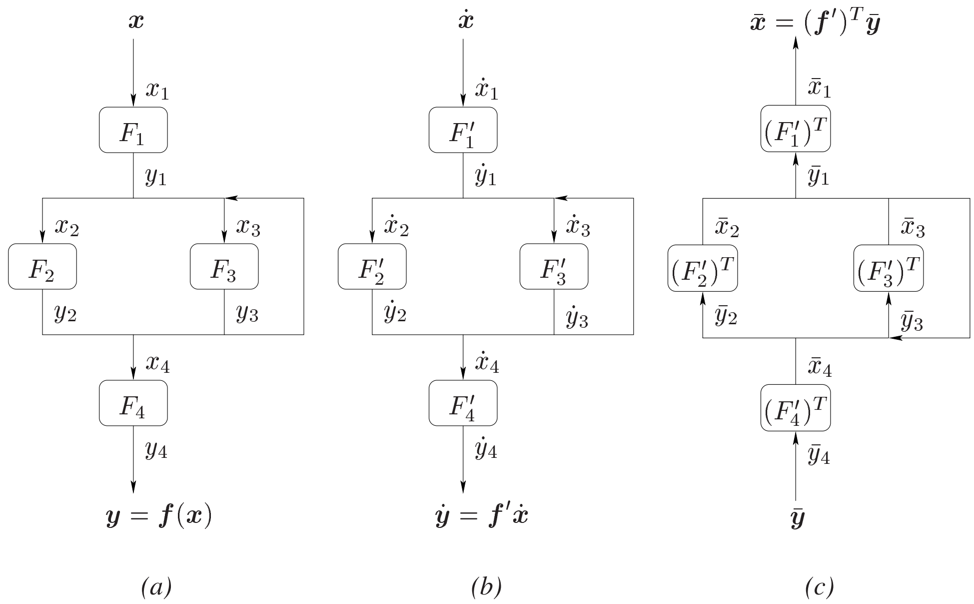

FIG. 2.1. *Control flow graph of (a) original code, (b) tangent linear model, (c) adjoint model*

If $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ is computed by a sequence of basic blocks $F_1, \ldots, F_l$ and assuming the availability of the local Jacobians $F_1', \ldots, F_l'$ the forward mode of AD computes

$$\dot{\boldsymbol{y}}_j = F_j'\dot{\boldsymbol{x}}_j \quad \text{for } j = 1, \ldots, l \quad . \tag{2.2}$$

Reverse mode AD propagates adjoints backward through the program flow as

$$\bar{\boldsymbol{x}}_j = (F_j')^T\bar{\boldsymbol{y}}_j \quad \text{for } j = l, \ldots, 1 \quad , \tag{2.3}$$

where $\boldsymbol{x}_j = (x_i^j : i = 1, \ldots, n_j)$ and $\boldsymbol{y}_j = (y_i^j : i = 1, \ldots, m_j)$ are the inputs and outputs of F_j, respectively. Figure 2.1 shows the resulting control flow graphs for a simple example involving a branch containing a loop. As pointed out in [18], the cost of the computation of the Jacobian of $\boldsymbol{f}$ is proportional to the number of edges $|E|$ in the DAG (see next section). Preaccumulation aims to reduce this number substantially to a number less than or equal to $n_j m_j << |E|$ at a comparatively low cost. Hence the propagation of bundles of directional derivatives (2.2) or adjoints (2.3) by vector forward or reverse mode [9], respectively, can become significantly more efficient.

2.2. Elimination Methods. Let $\boldsymbol{f}$ represent a basic block that is subject to preaccumulation as outlined in the previous section. The DAG $G = (V, E)$ is induced by the code for $\boldsymbol{f}$ [1]. Following the numbering in the previous section we have with the intermediate variables $v_1, \ldots, v_p \in Z$ the DAG's vertex set $V = X \cup Z \cup Y$. The numbering is subject to the dependence relation $\prec$, where $v_i \prec v_j$ (and $v_h \prec v_j$ as in (2.1)), and $v_i \prec v_j \Rightarrow i < j$. In Figure 2.2 (a) we show the DAG for the *code list* [9]

$$\begin{aligned} &v_1 = v_{-1} + v_0;\ v_2 = \sin(v_0);\ v_3 = v_1 + v_2;\ v_4 = v_1 * v_3; \\ &v_5 = \sqrt{v_3};\ v_6 = \cos(v_4);\ v_7 = -v_5 \quad . \end{aligned} \tag{2.4}$$

ETNA
Kent State University
etna@mcs.kent.edu

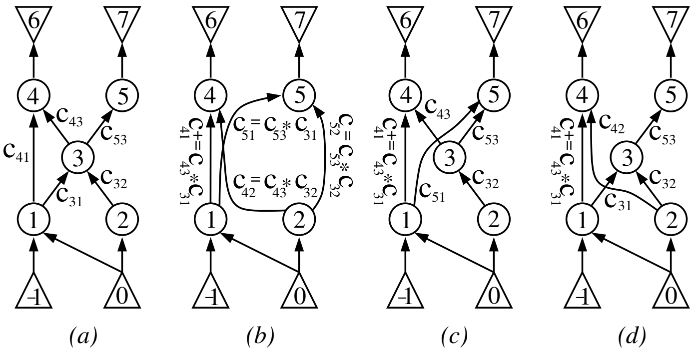

FIG. 2.2. *(a) Computational graph G for (2.4), (b) elimination of vertex* 3 *from G, (c) front elimination of edge* $(1, 3)$ *from G, (d) back elimination of edge* $(3, 4)$ *from G*

The intrinsics and operators provided by the underlying programming language constitute the possible elemental operations. Edges $(i, j) \in E$ are labeled with partial derivatives $c_{ji} = \frac{\partial \varphi_j}{\partial v_i} \in \mathbb{R}$ of the elemental operations associated with vertex j with respect to the corresponding arguments. For instance, in the example we have $c_{64} = -\sin(v_4)$.

Summarizing results from [11] and [18] we are looking for an elimination sequence σ that transforms G into a bipartite graph $\sigma(G)$ whose edge labels are the nonzero elements of $\boldsymbol{f}'$. The graph-based elimination steps are categorized in vertex, edge, and face eliminations. In G a vertex $j \in V$ is eliminated by connecting its predecessors with its successors. An edge (i, k) with $i \prec j$ and $j \prec k$ is labeled with $c_{ki} + c_{kj} * c_{ji}$ if it existed before the elimination of j. We say that *absorption* takes place. Otherwise, (i, k) is generated as *fill-in* and labeled with $c_{kj} * c_{ji}$ The vertex j is removed from G together with all incident edges. Figure 2.2 (b) shows the result of eliminating vertex 3 from the graph in Figure 2.2 (a).

An edge (i, j) is *front eliminated* by connecting i with all successors of j, followed by removing (i, j) [15]. The corresponding structural modifications of the DAG in Figure 2.2 (a) are shown in Figure 2.2 (c) for front elimination of $(1, 3)$. The new edge labels are given as well. Front elimination of edges eventually leads to intermediate vertices in G becoming *isolated*; that is, these vertices no longer have predecessors. Isolated vertices are simply removed from G together with all incident edges.

Back elimination of an edge $(i, j) \in E$ results in connecting all predecessors of i with j [15]. The edge (i, j) itself is removed from G. The back elimination of $(3, 4)$ from the graph in Figure 2.2 (a) is illustrated in Figure 2.2 (d). Again, vertices can become isolated as a result of edge-back elimination because they no longer have successors. Such vertices are removed from G.

Numerically the elimination is the application of the chain rule, that is, a sequence of *fused-multiply-add* (fma) operations

$$c_{ki} = c_{kj} * c_{ji} \quad (+c_{ki}) \quad \leftarrow \text{optional} \tag{2.5}$$

where the additions take place in the case of absorption or fill-in is created as described above. Aside from special cases a single vertex or edge elimination will result in more than one fma.

Face elimination was introduced as the elimination operation with the finest granularity of exactly one multiplication[2] per elimination step. Vertex and edge elimination steps have an interpretation in terms of vertices and edges of G, whereas face elimination is performed

[2] Additions are not necessarily directly coupled.

on the corresponding directed line graph $\mathcal{G}$. Following [18], we define the directed line graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ corresponding to $G = (V, E)$ as follows:

$$\begin{aligned}
\mathcal{V} = &\{ (i,j) : (i,j) \in E\} \cup \{ (\oplus,j) : v_j \in X\} \cup \{ (i,\ominus) : v_i \in Y\} \quad \text{and} \\
\mathcal{E} = &\quad \{((i,j), (j,k)) : (i,j),(j,k) \in E\} \\
&\cup \{((\oplus,j), (j,k)) : v_j \in X \wedge (j,k) \in E\} \\
&\cup \{((i,j), (j,\ominus)) : v_j \in Y \wedge (i,j) \in E\} \quad .
\end{aligned}$$

That is, we add a source vertex $\oplus$ and a sink vertex $\ominus$ to G connecting all independents to $\oplus$ and all dependents to $\ominus$. $\mathcal{G}$ has a vertex $v \in \mathcal{V}$ for each edge in the extended G, and $\mathcal{G}$ has an edge $e \in \mathcal{E}$ for each pair of adjacent edges in G. Figure 2.3 gives an example of constructing the directed line graph in (b) from the graph in (a). All intermediate vertices $(i,j) \in \mathcal{V}$ inherit the labels c_{ji}. In order to formalize face elimination, it is advantageous to move away from the double-index notation and use one that is based on a topological enumeration of the edges in G. Hence, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ becomes a DAG with $\mathcal{V} \subset I\!N$ and $\mathcal{E} \subset I\!N \times I\!N$ and certain special properties. The set of all predecessors of $j \in \mathcal{V}$ is denoted as P_j. Similarly, S_j denotes the set of its successors in $\mathcal{G}$. A vertex $j \in \mathcal{V}$ is called *isolated* if either $P_j = \emptyset$ or $S_j = \emptyset$. Face elimination is defined in [18] between two incident intermediate vertices i and j in $\mathcal{G}$ as follows:

1. If there exists a vertex $k \in \mathcal{V}$ such that $P_k = P_i$ and $S_k = S_j$, then set $c_k = c_k + c_j c_i$ *(absorption)*; else $\mathcal{V} = \mathcal{V} \cup \{k'\}$ with a new vertex k' such that $P_{k'} = P_i$ and $S_{k'} = S_j$ *(fill-in)* and labeled with $c_{k'} = c_j c_i$.
2. Remove (i,j) from $\mathcal{E}$.
3. Remove $i \in \mathcal{V}$ if it is isolated. Otherwise, if there exists a vertex $i' \in \mathcal{V}$ such that $P_{i'} = P_i$ and $S_{i'} = S_i$, then
   - set $c_i = c_i + c_{i'}$ *(merge)*;
   - remove i'.
4. Repeat Step 3 for $j \in \mathcal{V}$.

Figure 2.3 (c) shows the elimination of $(i,j) \in \mathcal{E}$, where $i = (1,3)$ and $j = (3,4)$. A complete face elimination sequence σ_f yields a tripartite directed line graph $\sigma_f(\mathcal{G})$ that can be transformed back into the bipartite graph representing the Jacobian $\boldsymbol{f}'$.

In [18] it was shown that vertex and edge eliminations can be interpreted as groups of face eliminations and that face elimination sequences can undercut the number of multiplications of an optimal vertex or edge elimination sequence. We note that any G can be transformed into the corresponding $\mathcal{G}$ but that a back transformation generally is not possible once face elimination steps have been applied. Therefore, face eliminations cannot precede vertex and edge eliminations.

A practical measure for the cost of computing $\boldsymbol{f}' = \sigma(G)$ is the count of multiplications $\#_*$ of edge labels. In Section 5 we discuss other options for measures of the computational cost.

3. Linearities and Constant Folding. The edge label multiplications can be categorized into *trivial*, *constant*, and *variable* multiplications ($*_t$, $*_c$, $*_v$) based on the type of operands as shown in Figure 3.1. For a repeated Jacobian computation (that is, computations for distinct $\boldsymbol{x}$), only the $*_v$ have to be re-executed. The $*_t$ can be transformed into toggling the sign, and $*_c$ can be executed at compile time in *constant propagation* fashion. Therefore we take only $\#_{*_v}$ as our cost measure. No known algorithm produces an optimal elimination sequence for a general DAG with polynomial complexity. To approximate an optimal $\hat{\sigma}_f(\mathcal{G})$

ETNA
Kent State University
etna@mcs.kent.edu

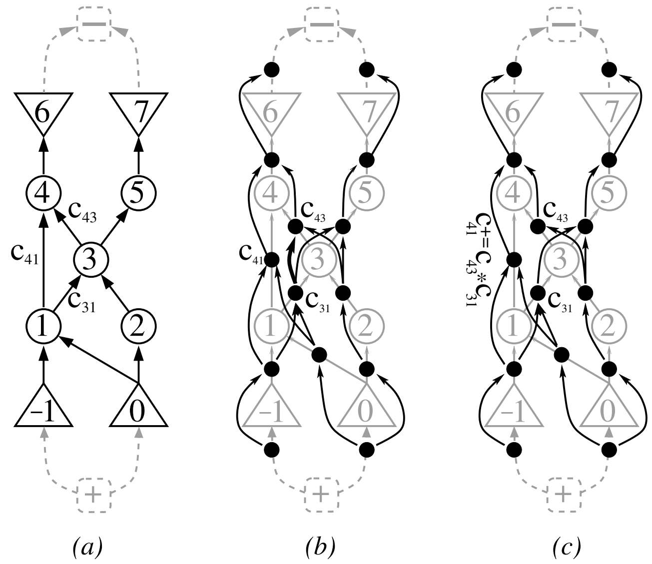

FIG. 2.3. *(a) G extended, (b) $\mathcal{G}$ overlaid, (c) face elimination*

such that

$$\#_{*_v}(\hat{\sigma}_f(\mathcal{G})) = \min_{\sigma_f(\mathcal{G})} \{\#_{*_v}(\sigma_f)\} \quad ,$$

we can use heuristics [19].

c_{ji} \ c_{kj}		trivial	constant	variable
trivial:	$c_{ji} \equiv \pm 1$	$*_t$	$*_t$	$*_t$
constant:	$c_{ji} \equiv const$	$*_t$	$*_c$	$*_v$
variable:	$c_{ji} = c_{ji}(\boldsymbol{x})$	$*_t$	$*_v$	$*_v$

FIG. 3.1. *Categories of multiplications $c_{kj} * c_{ji}$*

One can argue that code optimization via constant propagation and constant folding algorithms built into compilers is already capable of optimizing $*_t$ and $*_c$ away. Therefore, an AD source transformation tool would not necessarily have to be concerned with the explicit removal of the nonvariable multiplications. However, we need to be concerned with the cost of the heuristic approximation of $\hat{\sigma}$, which makes any reduction of the initial problem size by constant folding desirable, even though the approximation time is absorbed into compile time, not run time. For sufficiently large problems this cost is a critical hurdle, as is particularly evident for face elimination. The directed line graph $\mathcal{G}$ is a much larger data structure than G, which translates into a vast search space for any face elimination heuristic. A heuristic that is aware of the edge label categories and maintains those correctly for fill-ins and updates is even more complex and costly. Therefore, a viable heuristic may not distinguish label categories, and one can easily construct cases where $\#_*(\sigma_1) = \#_*(\sigma_2)$ but $\#_{*_v}(\sigma_1) < \#_{*_v}(\sigma_2)$. Constant folding can shrink the size of G (and $\mathcal{G}$) significantly for codes with large linear portions. We concentrate on the following issues:

ETNA
Kent State University
etna@mcs.kent.edu

1. Reduction of the problem size through constant folding in G
2. Constant folding and preservation of optimality
3. Implementation

As a starting point we consider a transformation of $G = (V, E)$ using a sequence σ_{e_c} of constant edge eliminations,[3] that is, $\#_{*_v}(\sigma_{e_c}(G)) = 0$. This yields $G' = (V', E') = \sigma_{e_c}(G)$ such that $|E'| \leq |E|$, which reduces the search space. An *optimality preserving* σ_{e_c} satisfies $\#_{*_v}(\hat{\sigma}_f(\mathcal{G}')) \not> \#_{*_v}(\hat{\sigma}_f(\mathcal{G}))$.

3.1. Single Expression Use Graphs. Similarly to Section 2.2 we denote the set of direct successors of a vertex v_i by $S_i = \{v_j | (i,j) \in E\}$; the set of direct predecessors of v_j is denoted by $P_j = \{v_i | (i,j) \in E\}$. G has the single expression use (seu) property if $|S_i| = 1\ \forall v_i \in Z$. This is true, for instance, for any set of right-hand sides of assignments that can be computed independently from each other. For such G there exists a polynomial algorithm that constructs an optimal elimination sequence $\hat{\sigma}_f$ first introduced in [17]. It uses the notion of an $X - v$ separating set defined as a set of vertices whose removal from G satisfies the condition that there be no path from any vertex in X to v. The separating set may contain a subset of X.

ALGORITHM 1 (optimal seu elimination). *For a given seu graph G perform vertex elimination steps in the following order, $\forall v_i, i = 1, \ldots, p$:*

(1) Find a minimal $X - v_i$ separating set $\underline{P_i}$.

(2) If $|\underline{P_i}| < |P_i|$ perform vertex elimination of all vertices $\{v_j : v_j \prec v_i \wedge v_k \prec v_j \forall v_k \in \underline{P_i}\}$ reverse ordered by index j.

After these steps have been performed for all v_i,

(3) perform vertex elimination of all remaining vertices reverse ordered by index j.

If G is a tree, then this algorithm yields the reverse mode of AD. Because of the seu property there is no advantage to be gained from face elimination over edge or vertex elimination; that is, $\hat{\sigma}_f$ can be written as $\hat{\sigma}_e$. The construction of $\hat{\sigma}_e$ attains the lower bound of operations

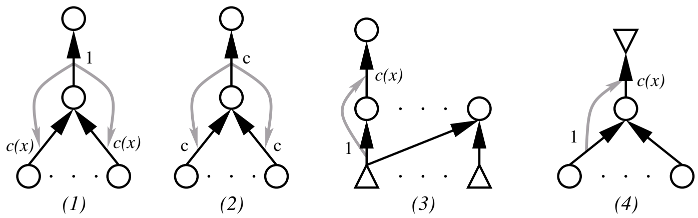

FIG. 3.2. *Four steps in Algorithm 2; labels* c *are constant, and* $c(x)$ *are variable*

required for each $v \in Z$ which can be defined by the size of the smallest $X - v$ separating vertex sets. The following algorithm creates a σ_{e_c} that reduces (steps 1-3) or maintains (step 4) the edge count; see also Figure 3.2.

ALGORITHM 2 (seu constant folding). *For a given G create σ_{e_c} with the following steps in order:*

(1) Back eliminate all trivial edges.

(2) Back eliminate (j,k) if (j,k) and (i,j) are constant $\forall i \in P_j$.

(3) Front eliminate all trivial edges (i,j) if $|P_j| = 1$

(4) Front eliminate all trivial edges (i,j) if $S_j \subseteq Y$.

One can easily see that none of the steps suggested here increases the minimal separating vertex set size and therefore σ_{e_c} is optimality preserving. Note that this may leave trivial and

[3] View a vertex elimination as a group of edge eliminations.

ETNA
Kent State University
etna@mcs.kent.edu

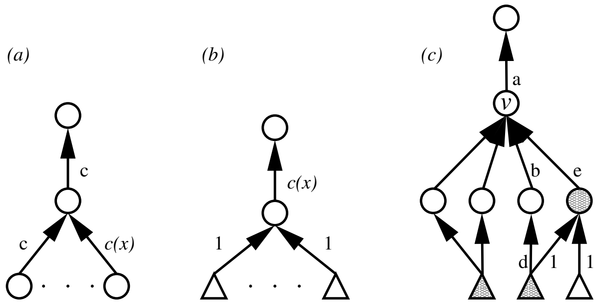

FIG. 3.3. *(a) and (b) situations not covered by Algorithm 2, (c) choosing a $X - v$ separating set*

constant edges in G' that cannot be eliminated with any of the steps given in this algorithm; see Figure 3.3 (a,b). While step 4 does not reduce the edge count, it is the result of considering the possibility of executing elimination steps that necessarily have to be part of the optimal elimination sequence. The application of the constant folding to seu graphs is a theoretical exercise as the optimal solution is constructed and does not require a search space reduction. However, Figure 3.3 (c) illustrates a case not covered by the purely structural information used in the construction of the optimal elimination sequence. Considering step 1 in Algorithm 1 there is a choice in picking a minimal separating set that may lead to a suboptimal elimination sequence. The $X - v$ separating set indicated by the gray filled vertices in Figure 3.3 (c) yields

$$t_1 = b * d; t_2 = a * c; t_3 = a * t_1; t_4 = 1 * t_2; t_5 = t_2 + t_4; t_6 = 1 * t_2$$

as part of the computation, whereas picking all independents as separating set yields

$$t_1 = b * d; t_2 = 1 * c; t_3 = 1 * c + t_1; t_4 = a * t_3; t_5 = a * t_2 \quad .$$

If, however, a, b, d are all constant and c is not, we have two versus one constant multiplications. Fortunately, an addition of minimal vertices is the only case exhibiting such a problem, and it can be overcome simply by choosing the $\underline{P_i}$ such that it leaves the elimination of the vertex in question for step 3.

3.2. Directed Acyclic Graphs. In Section 3 we mentioned the lack of an algorithm that exactly determines $\hat{\sigma}(G)$ with polynomial complexity. Algorithm 2 was motivated by the construction of $\hat{\sigma}_f$ in Algorithm 1 and the condition $|E'| \leq |E|$. The latter is the actual motivation here as the implied search space reduction permits computationally more expensive heuristics. Between seu graphs on the one side and generic DAGs on the other side there are currently no other, more generic structural properties of DAGs known to imply anything about the optimal elimination sequences. Therefore, a plausible starting point for generic DAGs is to require the seu property for subgraphs. We hypothesize that we preserve optimality through affecting the respective elimination subsequences only.

ALGORITHM 3 (DAG constant folding). *For a given G do the following steps in order:*

(1) If $\forall i \in P_j \wedge k \in S_j \; : \; c_{ji}, c_{kj}$ are constant, then eliminate[4] j if $|S_j| = 1$ or $|P_j| = 1$.
(2a) Back eliminate all trivial edges (i, j) if $|S_i| = 1$.
(2b) Front eliminate all trivial edges (i, j) if $|P_j| = 1$.

[4] interpreted as edge-front or edge-back elimination, respectively

ETNA
Kent State University
etna@mcs.kent.edu

Section 2.2 mentions the potential of face elimination in the corresponding directed line graph to undercut the operations count of vertex and edge eliminations. Therefore, we have to prove optimality preservation in terms of face elimination sequences.

PROPOSITION 1. *Back elimination of trivial edges with $|S_i| = 1$ (step 2a) preserves optimality.*

Proof. Assume $c_{ji} = 1$, and consider an optimal face elimination sequence σ for $\mathcal{G}$. We can construct σ' for $\mathcal{G}'$ the directed line graph for $G' = G - (i,j)$ with an iteration over σ. In each step k we construct a subsequence σ'_k and a remainder σ_{k+1}. Let r denote the vertex (i,j) in $\mathcal{G}$.

initialize:	$k := 1, \quad \sigma_1 := \sigma, \quad \mathcal{G}'_1 := \mathcal{G}', \quad \mathcal{G}_1 := \mathcal{G}$
while $\sigma_k \neq \emptyset$	split σ_k into $(X, (p,q), \sigma_{k+1})$,
	where (p,q) is the first face in σ_k with p or $q \in P_r \cup S_r$
	$\sigma'_k := X$
	if $(q \neq r) \wedge (p \neq r)$ then $\sigma'_k := \sigma'_k \cup (p,q)$ (*)
	$\mathcal{G}'_{k+1} := \sigma'_k(\mathcal{G}'_k), \quad \mathcal{G}_{k+1} := (X,(p,q))(\mathcal{G}_k)$ (apply eliminations)
	$k := k+1$

There are three scenarios for (*).

1. If $q \in S_r$ ($p \neq r$ is implied), then P_p and S_q are identical between $\mathcal{G}_k$ and $\mathcal{G}'_k$. That means the potential fill-ins are identical $\mathcal{F}_{\mathcal{G}_k}(p,q) = \mathcal{F}_{\mathcal{G}'_k}(p,q) = (\{v\}, \{(v,t) | t \in S_q\} \cup \{(s,v) | s \in P_p\})$; see Figure 3.4. Therefore, this case does not induce any further distinction between σ and σ'.

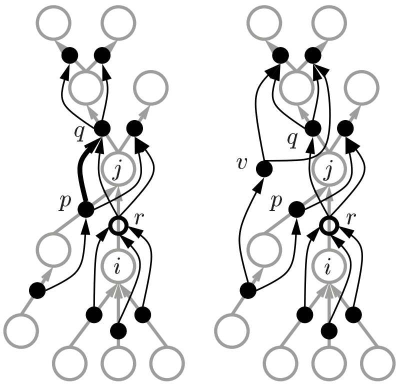

FIG. 3.4. *Scenario 1: a subgraph of $\mathcal{G}$ overlaid over the corresponding subgraph of G before (left) and after (right) face elimination of (p,q)*

2. If $q \in P_r$, then there may be a fill-in $\mathcal{F}_{\mathcal{G}_k}(p,q) = (\{v\}, \{(v,r)\} \cup \{(s,v) | s \in P_p\})$, which differs from $\mathcal{F}_{\mathcal{G}'_k}(p,q) = (\{v\}, \{(v,t) | t \in S_r\} \cup \{(s,v) | s \in P_p\})$. A subsequent elimination of (v,r) will be skipped according to the condition in (*). This coincides with the fact that the resulting edges are identical to the (v,t) that have already been created in $\mathcal{G}'_{k+1}$; see Figure 3.5. The sets $\{(s,v) | s \in P_q\}$ are identical, and subsequent eliminations of the (s,v) fall under (*).
3. The scenario of $p \in S_r$ is symmetric to $q \in P_r$.

The condition in (*) excludes some elimination steps (p,q). For the entire elimination sequence $\sigma' = \bigcup_k \sigma'_k$ we therefore have $|\sigma'| \not< |\sigma|$. □

The proof for the respective statement for $|P_j| = 1$ (step 2b) follows from symmetry.

ETNA
Kent State University
etna@mcs.kent.edu

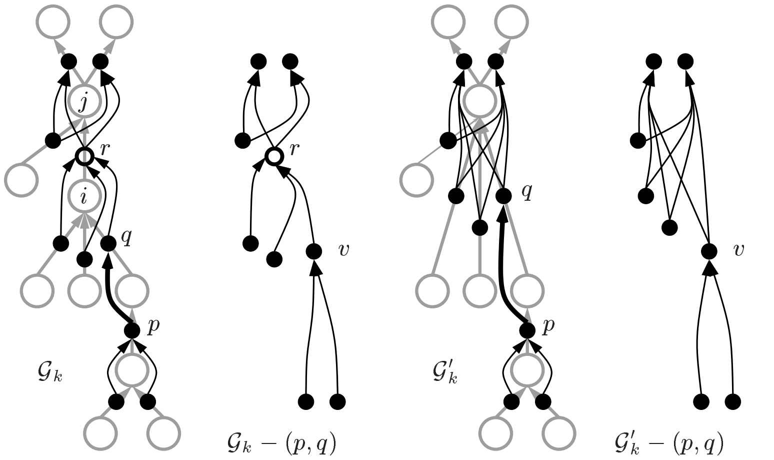

FIG. 3.5. *Scenario 2: face elimination before (left pair) and after (right pair) constant folding*

PROPOSITION 2. *Vertex elimination of j with (i, j) and (j, k) constant for all $i \in P_j$ and $k \in S_j$ and $(|S_j| = 1 \vee |P_j| = 1)$ preserves optimality (step 1)*

Proof. Assume the case with $|S_j| = 1$. We follow the same argument as the proof for Proposition 1 where we skip all elimination steps (p, q) for which $p = j,k$ or $q = j,k$. $|P_j| = 1$ follows from symmetry. □

All steps in Algorithm 3 reduce the edge count. Similarly to Algorithm 2 trivial and constant edges are left in the graph, and we can look for further reductions by adding the following steps to Algorithm 3:

(3a) Back eliminate all trivial edges (i, j) if $|P_i| = 1$.

(3b) Front eliminate all trivial edges (i, j) if $|S_j| = 1$.

This implies $|P_j| > 1$ and $|S_i| > 1$. Otherwise we would have used steps 2a or 2b, respectively.

PROPOSITION 3. *Front eliminating all trivial edges with $|S_j| = 1$ preserves optimality (step 3b).*

Proof. Again we use the same approach of skipping all elimination steps (p, q) with

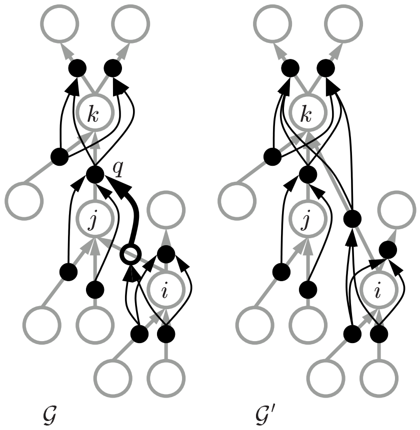

FIG. 3.6. *Trivial labels $c_{ji} = \pm 1$*

ETNA
Kent State University
etna@mcs.kent.edu

$p = (i,j)$ or $q = (i,j)$; see Figure 3.6. $\square$
Note that there is no reduction in the edge count unless $(i,k) \in G, S_j = \{k\}$, that is, we have absorption. In either case optimality is preserved. There are, however, two issues. First, even with absorption we have to account for the extra addition $c_{ki} = c_{ki} + c_{kj}$ when either c_{kj} or c_{ki} or both are variable; see also Section 5. Second, without absorption we duplicate the potentially variable label c_{kj} by removing the trivial label c_{ji}. This is contrary to the idea of preserving scarcity mentioned in Section 5. The proof for step 3a follows from symmetry.

Similar to steps 3a/3b we can also extend for constant labels with the following two steps:

(4a) Back eliminate (i,j) if $|P_j| = 1, P_i = \{h\}$ and $(h,i),(i,j)$ constant.

(4b) Front eliminate (i,j) if $|S_i| = 1, S_j = \{k\}$ and $(i,j),(j,k)$ constant.

Considering step 4b we observe that in case of absorption there will be no extra addition as long as (i,k) is constant. Even without absorption these steps preserve scarcity as there is no variable fill-in.

3.3. Constant Face Elimination. The suggested Algorithm 3 even with the extension steps 3 and 4 does not preeliminate all constant or trivial labels from G. Since we already pointed to the advantage face elimination may yield over vertex and edge elimination, we should consider the possibility of preeliminating constant faces. The search space for the directed line graph, is vastly larger than that for vertex or edge elimination. Still, the edge count in G, that is, the number of intermediate vertices in $\mathcal{G}$ is a reasonable, although crude, indicator for the search space size.

Any elimination of an (i,j) in $\mathcal{G}$ where $|S_i| = 1$ or $|P_j| = 1$ leads to the removal of i or j, respectively. It can be written as an edge elimination and would therefore already be covered by Algorithm 3. We consider scenarios that cannot be interpreted as edge eliminations. The least amount of structural change is therefore the removal of an edge in $\mathcal{G}$ by face elimination with absorption. That is, we consider an edge (i,j) with constant labels on i and j that has an absorbing vertex k (that is, $P_k = P_i$ and $S_k = S_j$) and k has a constant label as well. Consider the example in Figure 2.3. If both c_{31} and c_{43} are constant and none of the other edge labels are, then the previously introduced algorithm would not fold these two constants. One might expect that, similarly to the DAG, it was safe to preeliminate this face in the directed line graph because the result is absorbed in c_{41} and we basically just remove an edge from $\mathcal{G}$ at no cost. Somewhat surprising it turns out that such an edge removal may actually increase elimination cost, as the example in Figure 3.7 illustrates. The original graph has an optimal elimination sequence of length 3, for instance, $((2,4),(6,8),(3,7))$. Now we assume that both 3 and 7 have constant labels, and we eliminate $(3,7)$ and absorb into 5, thereby just removing $(3,7)$. One can perform an exhaustive search and see that there is no complete elimination sequence in the resulting graph $\mathcal{G} - (3,7)$ with a length ≤ 3.

An attempt to prove the preservation of optimality of a face elimination step that modifies $\mathcal{G}$ to $\mathcal{G}'$ would assume an optimal elimination sequence for $\mathcal{G}$ and then try to show that this elimination sequence contains a subsequence that is complete for $\mathcal{G}'$. This requires that all fill-in generated in $\mathcal{G}'$ is a subset of the fill-in generated in $\mathcal{G}$. With the current rule for merging of vertices in the face elimination definition, this is not necessarily the case once a single edge is removed.

4. Case Studies. The algorithms proposed in the previous section have been applied successfully to a large number of test problems. In Section 4.1 we introduce the format of the output graph generated by OpenAD[5] and we illustrate the effect of constant folding at the level of local partial derivatives. Two larger test cases are considered in Section 4.2 and Section 4.3. The linearized computational graphs are obtained by applying the flattening

[5] www.mcs.anl.gov/OpenAD

ETNA
Kent State University
etna@mcs.kent.edu

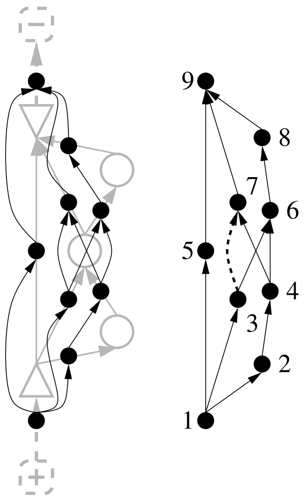

FIG. 3.7. *Removing* $(3, 7)$ *by constant face elimination*

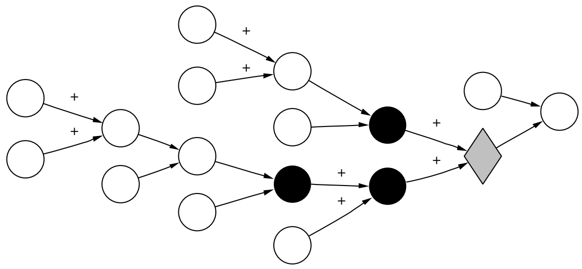

FIG. 4.1. *Oceanographic Box Model*

algorithms from [25] to the abstract internal representation of scalar assignments within OpenAD. The size of the graphs in terms of the number of intermediate vertices can often be reduced by up to 50% while decreasing the number of edges by about one third.

4.1. Oceanographic Box Model. The following single assignment is taken from a simplified box model [24] of the so-called 'thermohaline' circulation. This refers to the contribution to the ocean circulation which is driven by density gradients and thus controlled by temperature and salinity properties and its associated fluxes [5].

```
   dFldDt(1)= ( extForLoc(1)+gammaLoc*(fldStar(1)-fldNow(1))*vol(1)
+
uVelLoc*(fldNow(3)-fldNow(1))  )
      / vol(1)
```

A representation of the linearized DAG is shown in Figure 4.1. OpenAD uses the `graphviz` utility[6] to plot graphical representations of its internal representation. In the context of eliminating linearities the edge label '+' denotes local partial derivatives $c_{ji} \equiv \pm 1$. All the remaining variable c_{ji} do not have any label. Incidentally none of the examples used in this section exhibit any non trivial constant edge labels at the initial stage. In Figure 4.1, the diamond shaped vertex corresponds to the addition $w = u + v$ of the two subexpressions from the first and third line of the assignment statement above. According to Algorithm 2

[6] `www.graphviz.org`

ETNA
Kent State University
etna@mcs.kent.edu

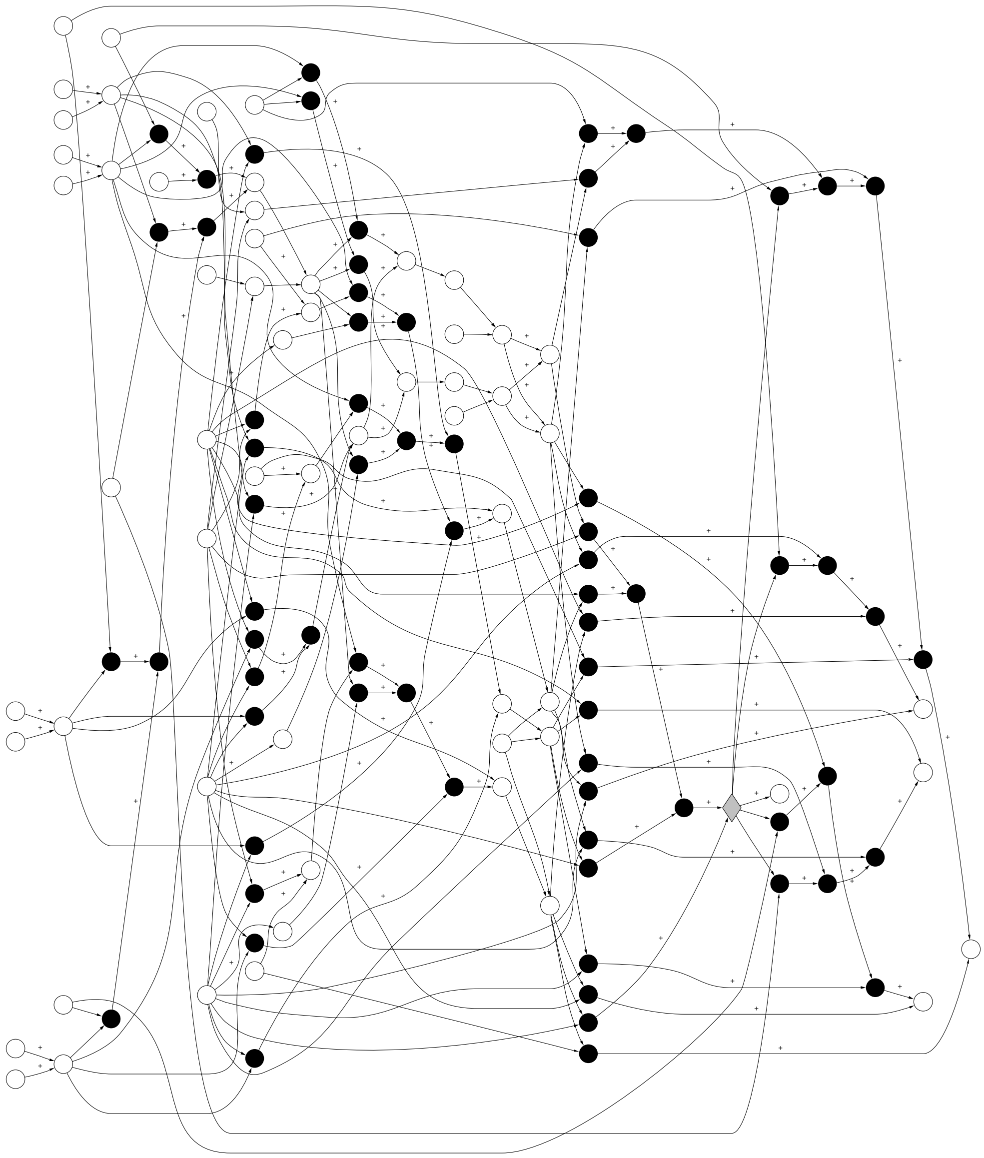

FIG. 4.2. *Roe Flux*

both predecessors of this vertex can be eliminated at compile time. They are blackened in Figure 4.1. For most applications the rules in Algorithm 2 are responsible for nearly all the reductions that are possible.

4.2. Roe Flux. Our second test problem defines the numerical fluxes of mass, energy, and momentum across a cell face in a finite volume compressible flow calculation. Roe's numerical flux [21] takes $n = 10$ inputs describing the flow on either side of a cell. It returns $m = 5$ outputs for the numerical flux. The 5×10 Jacobian is computed a large number of times during in the solution process justifying additional effort to optimize the accumulation procedure (see, for example, [22, 23, 8]).

ETNA
Kent State University
etna@mcs.kent.edu

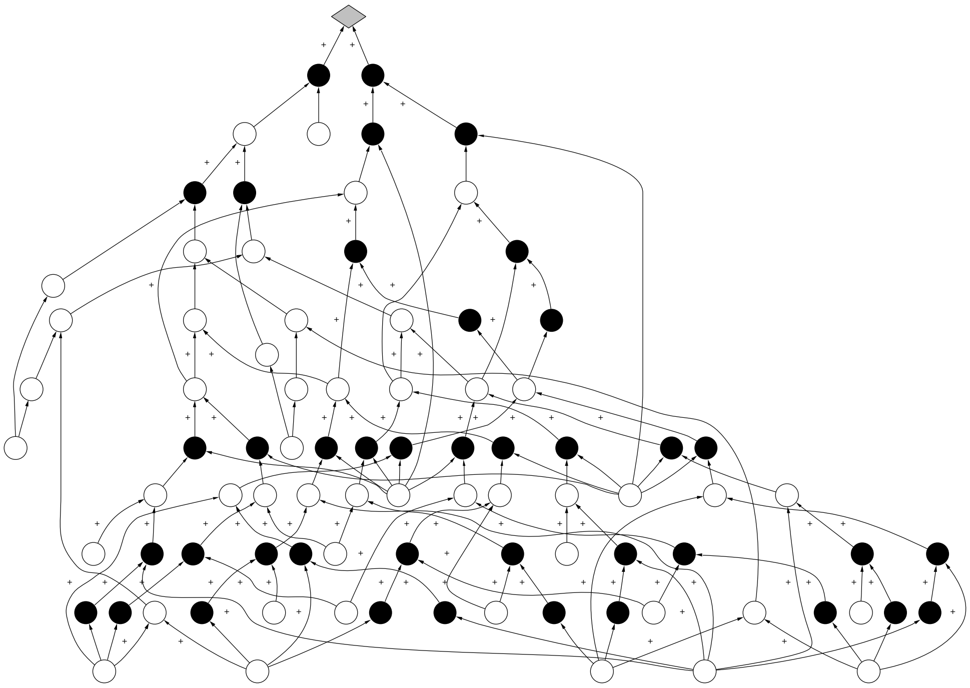

FIG. 4.3. *Flow in a Driven Cavity*

The biggest subgraph that could be obtained by the currently implemented version of the static flattening algorithm contains 131 vertices (20 minimal, 5 maximal). A graphical representation is shown in Figure 4.2. When relating the code fragment

```
  mu = alp15p / cave+nxhat * alp2 + nyhat * alp3 + nzhat * alp4
  dss1 = mu * thtave+ (cave * gm1inv * alp15p - uhat * alp15m)
+ utilde * alp2 + vtilde * alp3 + wtilde * alp4
  dss2 = mu
  dss3 = mu * uave-nxhat * alp15m + nzhat * alp3 - nyhat * alp4
  dss4 = mu * vave-nyhat * alp15m + nxhat * alp4 - nzhat * alp2
  dss5 = mu * wave-nzhat * alp15m + nyhat * alp2 - nxhat * alp3
```

to the graph one realizes that, for example, the diamond shaped vertex corresponds to mu. Both its predecessors can be eliminated.

In total 70 out of the 106 intermediate vertices can be eliminated at compile-time. All reductions are due to Algorithm 2 (1). Consequently, the number of edges is reduced by 70 too. The benefit for the runtime of local heuristics such as Lowest-Markowitz [11] or others [14, 2] is obvious. The savings become even more significant in the context of costly combinatorial optimization methods such as simulated annealing [16]. The number of configurations that can be checked is increased by more than a factor of two.

4.3. Flow in a Driven Cavity. The third test problem is from the MINPACK-2 test problem collection [3]. The 2D flow in a driven cavity is formulated as a boundary value problem, which is discretized by standard finite difference approximations to obtain a system of nonlinear equations. The value of the flow is computed at each grid point as

```
fvec(k) = (prlap-two*plap+pllap)/hx2 +
  (ptlap-two*plap + pblap)/hy2 -
  r*(dpdy*(prlap-pllap)/(two*hx)-dpdx*(ptlap-pblap)/(two*hy)) .
```

ETNA
Kent State University
etna@mcs.kent.edu

This value is represented by the maximal (diamond shaped) vertex in the graphical view of the linearized DAG shown in Figure 4.3. Obviously, both predecessors of this vertex can be eliminated at compile-time as arguments of a subtraction operation within an seu-subgraph. The original total number of 70 intermediate vertices can be reduced by 40. Again, the number of edges is reduced by the same amount.

5. Conclusion: Practical Use and Further Observations. Constant folding as presented here is implemented in the OpenAD[7] framework of the Adjoint Compiler Technology & Standards (ACTS) project. We mentioned in Section 1 that heuristics may be unaware of the label categories, as is the case with the face elimination heuristics currently used in OpenAD. The target application of OpenAD within the ACTS project is the MIT general circulation model. Since large portions of the model code are linear, one has a convincing case for using constant folding on G and creating a directed line graph only for the nonlinear core.

The choice of the number of multiplications as an optimality measure ignoring additions and memory access appears rather arbitrary when considering the raw execution time of a given Jacobian computation. We are well aware of the impact of data locality, pipelined operations, and so forth, since minimizing the execution time is the ultimate goal. OpenAD contains heuristics that address these practical aspects. For theoretical investigations it is certainly possible to count the individual multiplications and additions separately, as well as memory reads and writes. Additions occur optionally in conjunction with multiplications for vertex and edge elimination. For face elimination, however, one can easily construct cases where a single elimination step entails more than one addition. This is due to the current face elimination merge rule, where the elimination of a face enables the merging of up to two additional vertex pairs in $\mathcal{G}$. Most results on face elimination optimality ignore these additions altogether. It has been conjectured, however, that there is always an optimal elimination sequence that completely avoids additions through merging. This is subject of ongoing research. We also mentioned the issue of extra additions possibly introduced by steps 3 and 4 of Algorithm 3. In practice there is a principal dominance of the execution time of a multiplication over an addition which makes ignoring additions plausible. For data read and write operations there is no such generic statement and their execution times are highly hardware and context dependent. Including these timings would make general assumptions and optimality statements impossible. Moreover, the generated elimination code is itself subject to subsequent compiler optimization. Therefore, we consider the suggested optimality measure sufficient for this compiler and hardware-independent optimization.

In Algorithm 3, step 3, we mentioned the issue of scarcity preservation. There are functions $\boldsymbol{f}$ that have a dense $\boldsymbol{f}'$ but have graph representations with far fewer edges than the final bipartite graph. In [12, 10] the term *scarcity* was introduced to denote this property. Scarcity-preserving eliminations have the narrower objective of reducing or maintaining the number of edges with nontrivial labels. Despite the similarity to the objective of constant folding, there are some differences. With constant folding we eventually want to minimize operations for the computation of the Jacobian, whereas scarcity-preserving eliminations minimize the operations for repeated Jacobian vector products. A σ considered here for constant folding is complete, whereas a scarcity-preserving σ will generally be an incomplete elimination sequence. Moreover, the set of scarcity-preserving graph modifications suggested in [12, 10] contains a rerouting operation that can be interpreted as an inverse face elimination. This prevents an easy integration of both objectives. However, one can exploit the idea of cutting an elimination sequence short if there is an intermediate directed line graph representation with fewer vertices than the final tripartite directed line graph. Therefore, we note that the

[7] `www.mcs.anl.gov/OpenAD`

proposed constant folding steps 1, 2a/2b and 4a/4b in Algorithm 3 preserve scarcity as well. Steps 3a/3b preserve scarcity only for absorbed fill-in and if the absorbing label is variable. We can modify the heuristics to represent label categories in the directed line graph to enable a scarcity-preserving face elimination. The investigation of such a modified optimization criterion is the subject of ongoing research.

Acknowledgments. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38, and by the National Science Foundation's Information Technology Research Program under Contract OCE-0205590, "Adjoint Compiler Technology & Standards" (ACTS). The implementation work was carried out in part by Peter Fine during his summer internship at Argonne's Mathematics and Computer Science Division in 2004.

REFERENCES

[1] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] A. ALBRECHT, P. GOTTSCHLING, AND U. NAUMANN, *Markowitz-type heuristics for computing Jacobian matrices efficiently*, in Computational Science – ICCS 2003, vol. 2658 of Lecture Notes in Computer Science, Springer, 2003, pp. 575–584.

[3] B. AVERIK, R. CARTER, AND J. MORÉ, *The MINPACK-2 test problem collection (preliminary version)*, Technical Memorandum ANL/MCS-TM-150, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[4] M. BERZ, C. BISCHOF, G. CORLISS, AND A. GRIEWANK, eds., *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996, SIAM.

[5] P. CLARK, N. PISIAS, T. STOCKER, AND A. WEAVER, *The role of the thermohaline circulation in abrubt climate change*, Nature, 415 (2002), pp. 863–869.

[6] G. CORLISS, C. FAURE, A. GRIEWANK, L. HASCOET, AND U. NAUMANN, eds., *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002, Springer.

[7] G. CORLISS AND A. GRIEWANK, eds., *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991, SIAM.

[8] S. FORTH, M. TADJOUDDINE, J. PRYCE, AND J. REID, *Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding*, ACM Trans. Math. Softw., 3 (2004), pp. 266–299.

[9] A. GRIEWANK, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, no. 19 in Frontiers in Appl. Math., SIAM, Philadelphia, 2000.

[10] A. GRIEWANK, *A mathematical view of automatic differentiation*, Acta Numerica, 12 (2003), pp. 321–398.

[11] A. GRIEWANK AND S. REESE, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 126–135.

[12] A. GRIEWANK AND O. VOGEL, *Analysis and exploitation of Jacobian scarcity*, in Proceedings of HPSC Hanoi, Springer, 2003, to appear.

[13] S. MUCHNICK, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, 1997.

[14] U. NAUMANN, *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*, PhD thesis, Technical University of Dresden, December 1999.

[15] ———, *Elimination techniques for cheap Jacobians*, in Automatic Differentiation of Algorithms: From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, eds., Computer and Information Science, Springer, New York, 2001, ch. 29, pp. 247–253.

[16] ———, *Cheaper Jacobians by Simulated Annealing*, SIAM J. Opt., 13 (2002), pp. 660–674.

[17] ———, *Optimal pivoting in tangent-linear and adjoint systems of nonlinear equations*, Preprint ANL-MCS/P944-0402, Argonne National Laboratory, 2002.

[18] ———, *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph*, Math. Prog., 99 (2004), pp. 399–421.

[19] U. NAUMANN AND P. GOTTSCHLING, *Simulated annealing for optimal pivot selection in Jacobian accumulation*, in Stochastic Algorithms, Foundations and Applications – SAGA'03, A. Albrecht, ed., Lecture Notes in Computer Science, Berlin, 2003, Springer, to appear.

[20] L. B. RALL, *Automatic Differentiation: Techniques and Applications*, vol. 120 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.

[21] P. ROE, *Approximating Riemann solvers, parameter vectors, and difference schemes*, J. Comp. Physics, 43 (1981), pp. 357–372.

[22] M. TADJOUDDINE, S. FORTH, AND J. PRYCE, *AD tools and prospects for optimal AD in CFD flux calculations*, in Automatic Differentiation of Algorithms – From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, eds., New York, 2002, Springer, pp. 255–261.

[23] M. TADJOUDDINE, S. FORTH, J. PRYCE, AND J. REID, *Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation*, in Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, eds., vol. 2330 of Lecture Notes in Computer Science, Berlin, 2002, Springer, pp. 1077–1086.

[24] E. TZIPERMAN AND P. IOANNOU, *Transient growth and optimal excitation of thermohaline variability*, J. Phys. Oceanogr., 32 (2002), pp. 3427–3435.

[25] J. UTKE, *Flattening basic blocks*, in Automatic Differentiation: Applications, Theory, and Tools, H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, eds., Lecture Notes in Computational Science and Engineering, Springer, 2005.