RWTH Aachen

Department of Computer Science
Technical Report

Optimal Vertex Elimination in Single-Expression-Use Graphs

Uwe Naumann and Yuxiao Hu

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2006-08

RWTH Aachen · Department of Computer Science · May 2006

Optimal Vertex Elimination in Single-Expression-Use Graphs

Uwe Naumann and Yuxiao Hu

LuFG Software and Tools for Computational Engineering, Department of Computer Science
RWTH Aachen University, 52056 Aachen, Germany
WWW: `http://www.stce.rwth-aachen.de`, Email: `naumann@stce.rwth-aachen.de`

Abstract. Gradients of scalar multivariate functions can be computed by elimination methods on the linearized computational graph. The combinatorial optimization problem that aims to minimize the number of arithmetic operations performed by the elimination algorithm is known to be NP-complete. In this paper we present a polynomial algorithm for solving a relevant subclass of this problem's instances. The proposed method relies on the ability to compute vertex covers in bipartite graphs in polynomial time. A simplified version of this graph algorithm is used in a research prototype of the differentiation-enabled NAGWare Fortran compiler for the preaccumulation of local gradients of scalar assignments in the context of automatic generation of efficient tangent-linear code for numerical programs.

1 Context

The motivation for the graph-algorithmic work in this paper has its origins in automatic differentiation (AD) of numerical programs. In AD we consider implementations of vector functions $F : I\!R^n \to I\!R^m$ as computer programs written in some (often high-level) imperative programming language. Whenever we talk about F we actually mean the given implementation.[1] For given input values we expect the program that computes the values of *dependent* output variables $\mathbf{y} = (y_k)_{k=1,\ldots,m}$ by $\mathbf{y} = F(\mathbf{x})$ as a function of *independent* input variables $\mathbf{x} = (x_i)_{i=1,\ldots,n}$ to decompose into a sequence of *elementary assignments*

$$v_j = \varphi_j(v_i)_{i \prec j}, \quad j = 1, \ldots, p+m \quad . \tag{1}$$

Equation (1) is also referred to as the *single assignment code.* The direct dependence of v_j on v_i is denoted by $i \prec j$. Hence, v_i is an argument of the *elemental function* φ_j.[2] The transitive closure of this relation is denoted by $\prec^*$. It is reflexive, that is $j \prec^* j$. We use p *intermediate* variables v_j, $j = 1, \ldots, p$, and we set $v_{i-n} = x_i$, $i = 1, \ldots, n$ and $v_{p+k} = y_k$, $k = 1, \ldots, m$. Whenever appropriate we use $q = p+m$ for notational brevity. We follow Griewank's notation [9] wherever possible. Refer to the same source for a comprehensive discussion of both fundamental and advanced issues in AD. Applications and special topics in past and ongoing research and development in the field are covered by the proceedings of the four international workshops on AD held in 1991 [6], 1996 [2], 2000 [5], and 2004 [4].

[1] Often there are multiple alternatives for implementing a given function. We put ourselves into the position of a compiler that "sees" the given implementation but not the underlying problem that the programmer had intended to implement.

[2] Think of elemental functions as the arithmetic operators and numerical intrinsics provided by your favorite programming language.

Equation (1) induces a directed acyclic graph (dag) $G = (V, E)$ (also called the *computational graph*) with $V = \{1-n, \ldots, q\}$ and $(i, j) \in E \Leftrightarrow i \prec j$. With the mutually disjoint[3] vertex sets X, Z, and Y representing the independent, intermediate, and dependent vertices, respectively, we also write

$$G = (\{X, Z, Y\}, E) \quad .$$

For given values of the inputs the dag is *linearized* by attaching for all i, j with $i \prec j$ the values of the *local partial derivatives*

$$c_{j,i} = c_{j,i}(v_k)_{k \prec j} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \tag{2}$$

to the corresponding edges (i, j). It has been well known for some time that the Jacobian matrix $F' = F'(\mathbf{x}) \in I\!R^{m \times n}$ can be accumulated by making G bipartite (transformation $G \to G'$ where $G' = (\{X, \varnothing, Y\}, E')$ by the elimination techniques to be discussed below) with the labels on the remaining edges in E' representing exactly the nonzero entries of F'.

The first elimination technique is due to Baur and Strassen [1] who interpret the computation of each Jacobian entry

$$F'(j, i) \equiv \frac{\partial y_j}{\partial x_i}(\mathbf{x}) = \sum_{\pi \in \{(1-i) \to (p+j)\}} \prod_{(k,l) \in \pi} c_{l,k} \tag{3}$$

as the elimination of all paths π connecting the corresponding vertices $1 - i$ and $p + j$ in G.[4] In 1991 Griewank and Reese [10] propose a vertex elimination technique to get from G to G' together with a local heuristic for approximating the minimal number of arithmetic operations required for the transformation. Refined versions in the form of edge and face elimination techniques are proposed by Naumann in 1999 [15] and 2004 [16]. Griewank and Vogel [11] observe that pure chain-rule-based arithmetic may not minimize the operations count in some situations. They propose a rerouting technique motivated by a kind of LU-factorization of a submatrix of the *extended Jacobian* [9].

In the following we review some of the known elimination techniques. We focus on the aspects that are essential for further understanding the development of this paper's ideas.

The application of Equation (3) to the graph in Figure 1 (a) yields the bipartite graph displayed in Figure 1 (b). The rule for eliminating an intermediate vertex j from G follows immediately. Vertex 1 is eliminated in Figure 1 at the cost of $|P_1| \cdot |S_1| = 2 \cdot 2 = 4$ scalar floating-point multiplications.[5] We use P_j and S_j to denote the sets of predecessors and successors of a vertex j in G, respectively. With $|A|$ denoting the cardinality of a set A, the indegree and outdegree

[3] W.l.o.g. we assume that dependent variables never occur as arguments of an elemental function.

[4] We use the notation $\{i \to j\}$ to denote the set of all distinct paths from i to j. Paths in G are regarded as ordered sets (sequences) of edges.

[5] For the time being we use the number of scalar floating-point multiplications as a cost function. Later we will also count scalar additions. We shall see that multiplications and additions are the only operations performed during the accumulation of Jacobians using the elimination techniques on the type of linearized computational graph that this paper focuses on.

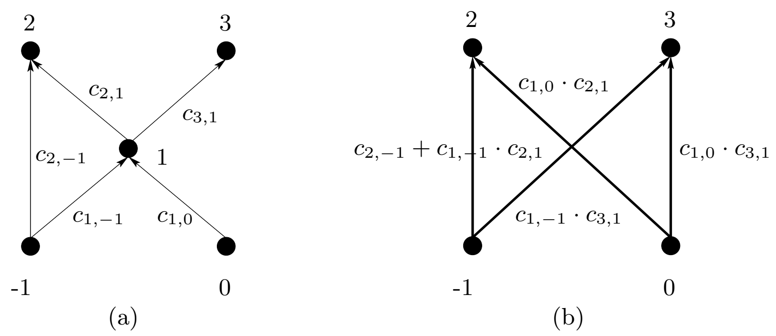


Fig. 1. Linearized Computational Graphs

of a vertex j are equal to $|P_j|$ and $|S_j|$, respectively. Applying Equation (3) to a single column or row in the 2×2 Jacobian leads to rules for *front-* and *back-*elimination of edges, respectively. For example, the first column is computed by front-elimination of $(-1, 1)$. Back-elimination of $(1, 3)$ yields the second row. Refer to Figure 2 for illustration. Naumann shows in [15] that there are graphs for which the cost (in terms of the number of scalar multiplications and additions) of the best edge elimination sequence undercuts that of the best vertex elimination sequence.

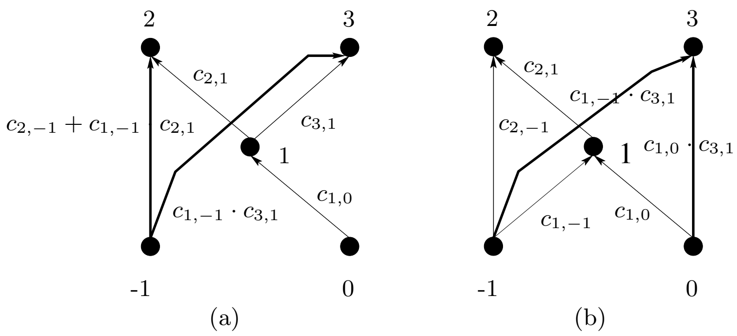


Fig. 2. Edge Elimination

For the graph in Figure 1 neither front- nor back-elimination allow for single Jacobian entries to be computed without affecting others. In [16] Naumann shows that this independent evaluation of partial derivatives is required to undercut the cost of the best edge elimination sequence. He introduces *face elimination* as an edge elimination technique on the *dual linearized computational graph* (a special form of the original graph's line graph). The step to the dual graph is required as there is no structural equivalent to face elimination in the original graph. Later we shall see that any face elimination sequence can be expressed by an equivalent edge elimination sequence for the type of graphs that this paper focuses on.

Rerouting has been introduced in [11] as an elimination technique that allows for the number of arithmetic operations to be lowered even further. Intuitively, the contribution of a local partial derivative in G (an edge label) is rerouted via an alternative path from the edge's source vertex to its target. In this paper we focus on pure chain rule arithmetic. Rerouting is not considered.

2 Single-Expression-Use Graphs

Definition 1 *A single-expression-use graph (SEU-graph) is a linearized computational graph whose intermediate vertices have outdegree one.*

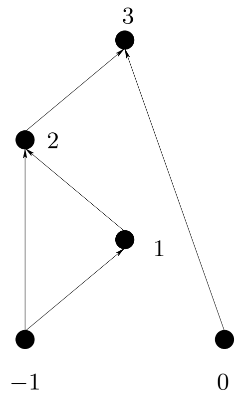


Fig. 3. Single-Expression-Use Graph

Typical representatives of SEU-graphs are induced by right-hand sides of scalar assignments that are frequently used in numerical programs.

Example. Consider

$$y = x_1 \cdot \sin(x_1)/x_2 \quad . \tag{4}$$

The single assignment code of Equation (4) is

$$v_1 = \sin(v_{-1});\ v_2 = v_{-1} \cdot v_1;\ v_3 = v_2/v_0 \quad .$$

It yields the computational graph in Figure 3. Systems of equations can yield SEU-graphs. For example, in

$$t = x_1 \cdot \sin(x_1); \quad y_1 = t/x_2; \quad y_2 = x_3/x_1$$

the computations of the two outputs y_1 and y_2 are mutually independent in that they do not share a common intermediate value. □

Larger SEU-graphs can result from sequences of scalar assignments provided that one is able to *flatten* the corresponding sequence of smaller SEU-graphs [20].

Example. The first two equations of the previous example

$$t = x_1 \cdot \sin(x_1); \quad y_1 = t/x_2$$

yield the SEU-graph in Figure 3. In order to be able to flatten the corresponding two separate SEU-graphs one needs to establish that the t on the left-hand side of the first assignment refers to the same memory location as the t on the right-hand side of the second assignment. This task is easy to accomplish if the variable is both static and scalar (as it is in this example). However, if the memory reference is dynamic (via pointers or array accesses with runtime-dependent indexes), then the unique identification is often impossible – undecidable in general. □

In this paper we present an algorithm that minimizes the number of arithmetic operations required for the accumulation of first derivatives of numerical programs whose computational graphs are SEU-graphs. We prove the correctness and optimality of an algorithm that uses vertex elimination [10] to achieve the transformation from the original dag into a bipartite one. The proof is based on the assumption that all local partial derivatives that are defined in Equation (2) are algebraically independent. Although this may be a serious restriction in theory (see [17]) we believe that for most practical situations the algorithm does produce optimal results, that is, derivative accumulation procedures with minimum arithmetic cost. Moreover it contributes to our still not satisfactory understanding of the structural properties of optimal derivative code.

3 Optimal Vertex Elimination in SEU-Graphs

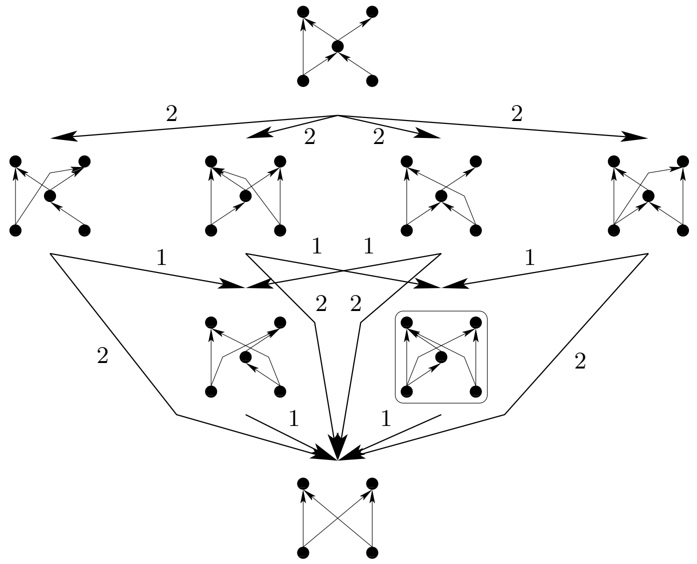


Fig. 4. Search Space for Optimal Edge Elimination Problem: The edges in this meta graph are labeled with the number of multiplications involved in the respective elimination step. We shall see later that only the number of multiplications is of interest from a combinatorial point of view as the number of additions is constant for SEU-graphs.

The main result of this section is a vertex elimination algorithm that minimizes the number of arithmetic operations required for the accumulation of first derivatives using elimination techniques on SEU-graphs. We have already seen that face elimination does not contribute to improving the best edge elimination sequence. Despite the fact that not every edge elimination sequence has an equivalent vertex elimination sequence[6] we shall see that vertex elimination is in fact sufficient.

According to Equation (3) the accumulation of the Jacobian involves the elimination of all intermediate vertices. We are looking for an edge elimination sequence that accomplishes this task at minimum cost. Note that the elimination of a vertex is equivalent to the back-elimination of its outgoing edges. There is only one outgoing edge per intermediate vertex in SEU-graphs. The front-elimination of all incoming edges has a similar effect. The search space of this combinatorial optimization problem consists of all graphs that can be generated by applying arbitrary edge elimination sequences to the original graph G. For example, Figure 4 shows the search space for the graph in Figure 1 (a). The solution of the optimal edge elimination problem is equivalent to that of a shortest path problem in the *meta graph* whose size is exponential in the size of the original graph G [16].[7] We need to front-[back-]eliminate a minimum of $|\underline{P}_j|$ [$|\underline{S}_j|$] in-[out-]edges to eliminate some vertex j. An immediate consequence of Equation (3) is that the sets $\underline{P}_j$ are minimal X-j vertex cuts in G, that is minimal vertex sets $C_v \subseteq X \cup Z \setminus \{j\}$ with the property that every path from some $i \in X$ to $j \in Z$ includes a vertex from C_v. Similarly, the sets $\underline{S}_j$ are minimal j-Y vertex cuts. Hence we get the following lower bound on the cost of Jacobian accumulation by edge elimination on the linearized computational graph.

Lemma 1. *The minimal number of multiplications required for the elimination of a vertex j over all possible edge elimination sequences in G is equal to $|\underline{P}_j||\underline{S}_j|$.*

Proof. According to Equation (3) the elimination of a path $(i \to j)$ makes i an immediate predecessor of j. The number of immediate predecessors of j cannot undercut that of a minimal vertex cut since by Menger's theorem [12] the latter is equal to the maximum number of vertex-disjoint paths connecting the vertices in X with j. ∎

Lemma 2. *The minimal number of multiplications required for the transformation $G \to G'$ is equal to $\sum_{j \in Z} |\underline{P}_j||\underline{S}_j|$.*

Proof. The proof follows immediately from Lemma 1. ∎

A *reverse vertex elimination sequence* eliminates all intermediate vertices such that for any two $i, j \in Z$ the vertex j is eliminated before i whenever $i \prec^* j$. We say that the vertices are reverse eliminated. With the outdegree of all intermediate vertices in SEU-graphs being equal to one we can prove the following result.

Lemma 3. *A reverse vertex elimination sequence minimizes the number of scalar multiplications for SEU-graphs whose intermediate vertices have minimum indegree, that is, $\forall j \in Z : |P_j| = |\underline{P}_j|$.*

Proof. According to Lemma 1 the minimum cost of accumulating the Jacobian of an SEU-graph is $\sum_{j \in Z} |\underline{P}_j|$. While the elimination of an intermediate vertex may change the indegree of its successor it leaves the outdegrees of its predecessors

[6] For example, in Figure 3 the front-elimination of edge $(1, 2)$ followed by the elimination of vertex 1 and vertex 2 has no equivalent in terms of vertex elimination sequences.

[7] It is easy to verify that the length of any path is equal to four in Figure 4.

constant (equal to one). Consequently, a reverse vertex elimination sequence eliminates all intermediate vertices at minimal cost. Hence, it reaches the lower bound established by Lemma 2. ■

Lemma 4. *All elimination sequences perform* $\sum_{i \in X} |S_i| - \mu$ *additions when applied to an SEU-graph* G. *The parameter* μ *denotes the number of nonzeros in the corresponding Jacobian* F'.

Proof. First we show that the number of additions performed in Equation (3) cannot be undercut for SEU-graphs. The proof is by deriving a contradiction.

Suppose that some elimination sequence decreases the number of additions. Then, by the distributive law, some sum

$$s = \prod_{(i_1,j_1)\in(i\to j)_1} c_{j_1,i_1} + \prod_{(i_2,j_2)\in(i\to j)_2} c_{j_2,i_2},$$

where $(i \to j)_\nu \in \{i \to j\}$, $\nu = 1, 2$, are two vertex disjoint paths that connect i and j, must appear as a common subterm in two or more Jacobian entries. The above implies that $i \in X$ since only independent vertices can have more than one successor. Moreover, j or some k with $j \prec^* k$ needs to have at least two successors for s to appear in two different Jacobian entries. The above yields a contradiction to the definition of SEU-graphs. Consequently, $|\{i \to j\}| - 1$ additions must be performed for each pair $i \in X$ and $j \in Y$.

The sum over all these pairs is equal to $\mathcal{N} - \mu$, where $\mathcal{N}$ denotes the number of all distinct paths in G. Let n_i denote the number of distinct paths through a vertex i in G. Furthermore, let n_i^b $[n_i^f]$ denote the number of different paths leading into [emanating from] a vertex i. Initializing $n_i^b = 1$ for $i \in X$ and $n_i^f = 1$ for $i \in Y$, we get

$$n_i^b = \sum_{j \in P_i} n_j^b, \quad \text{and} \quad n_i^f = \sum_{j \in S_i} n_j^f \quad . \tag{5}$$

It follows that

$$n_i = \begin{cases} n_i^b & i \in Y \\ n_i^f & i \in X \\ n_i^b n_i^f & i \in Z \end{cases} ,$$

and we get $\mathcal{N} = \sum_{j \in X} n_j = \sum_{j \in Y} n_j$. By the definition of SEU-graphs $n_i^f = 1$ for all $i \in Z$. With Equation (5) it follows that

$$\mathcal{N} = \sum_{j \in Y} n_j = \sum_{j \in Y} n_j^b = \sum_{i \in X} n_i^f = \sum_{i \in X} |S_i| \quad .$$

The proof of the lemma follows immediately. ■

Example. The application of the above argument to the graph in Figure 3 delivers the following.

We set $n_{-1}^b = n_0^b = 1$. It follows that

1. $n_1^b = n_{-1}^b = 1$
2. $n_2^b = n_1^b + n_{-1}^b = 2$
3. $n_3^b = n_2^b + n_0^b = 3$.

The gradient to be computed contains two entries. Hence, $1 = 3 - 2$ addition is performed by any elimination sequence. □

Theorem 2 states an optimal vertex elimination algorithm for SEU-graphs that minimizes the number of multiplications by reaching the lower bound established in Lemma 2. The algorithm relies on the ability to compute the cardinality of minimal X-j vertex cuts for all $j \in Z$ as well as a corresponding minimizer.

For any $j \in Z$ we consider the single-source-single-sink dag

$$\begin{aligned} \dot{G}_j &= (\dot{V}_j, \dot{E}_j) \\ \dot{V}_j &= \{i \in V : i \prec^* j\} \cup \{s\} \\ \dot{E}_j &= \{(i,k) \in E : i, k \in \dot{V}_j\} \cup \{(s,i) : i \in X \cap \dot{V}_j\} \end{aligned} \tag{6}$$

as a flow network with integer edge capacities $c(i,k) = 1$ for all $(i,k) \in \dot{E}_j$. Negative flows are disallowed. Hence, the flow $\phi(i,k)$ along any edge $(i,k) \in \dot{E}_j$ can be either 0 or 1. According to the Maximum-Flow-Minimum-Cut theorem [8] the cardinality of a minimal edge cut separating vertex j from s is equal to a maximal flow from s to j in $\dot{G}$. A maximal flow from s to j can be computed by the well-known Ford-Fulkerson algorithm [8] via the gradual saturation of all augmenting paths ensuring that all paths from s to j have been eliminated in the corresponding residual network. The complexity of this algorithm is known to be $O(|\dot{E}_j| \cdot \bar{\Phi})$ where $\bar{\Phi}$ denotes the value of a maximal flow in $\dot{G}_j$. Motivated by the worst-case complexity of the Ford-Fulkerson algorithm that is known to be exponential in the size of the underlying network Edmonds and Karp [7] presented an improvement of the algorithm by implementing the computation of the augmenting path with a breadth-first search. This approach ensures a polynomial complexity of $O(|\dot{E}_j| \cdot |\dot{V}_j|^2)$. For networks based on SEU-graphs as defined above we observe that $\bar{\Phi} \leq |P_j|$ for all $j \in Z$. Hence, the complexity of the Ford-Fulkerson algorithm is $O(|\dot{E}_j| \cdot |\dot{V}_j|)$.

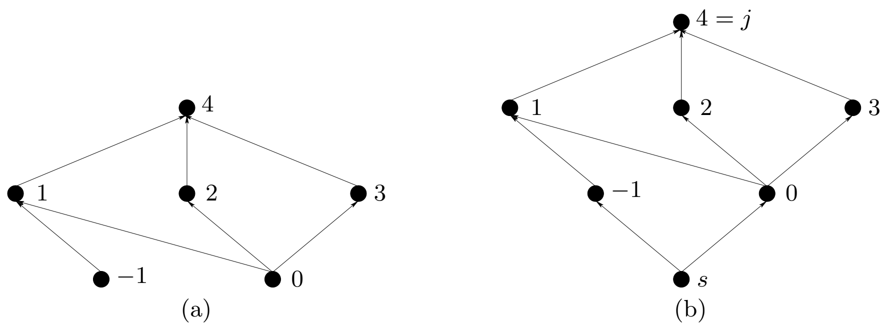


Fig. 5. G and $\dot{G}_4$

Example. $\dot{G}_4$ is shown in Figure 5 (b) for the graph in Figure 5 (a). □

We partition the vertices in $\dot{G}_j$ as

$$\dot{V}_j = (s, X_j, Z_j, P_j, j) \quad , \tag{7}$$

where s is the unique source, j is the unique sink, $X_j = S_s$ are those independent vertices that are connected to j by some path, P_j are the immediate predecessors of j, Z_j are the remaining vertices. We require $Z_j \cap (X_j \cup P_j) = \varnothing$.

Note that $\forall i \in X_j : |P_i| = 1$ and $\forall j \in Z_j \cup P_j \setminus X_j : |S_j| = 1$ in $\dot{G}_j$. We define a s-j edge cut as a set C_e of edges with the property that every path from s to j contains an edge from C_e.

Lemma 5. *Let C_e be a minimal s-j edge cut in $\dot{G}_j$. Then*

$$\exists C'_e : \big(\forall (i,k) \in C'_e \;\; i \notin X_j\big) \wedge |C'_e| = |C_e| \quad .$$

Proof. The lemma states that there is always a minimal s-j edge cut in $\dot{G}_j$ such that none of its elements emanates from a vertex in X_j.

Let $(i,k) \in C_e$ where $i \in X_j$. Then $C'_e = (C_e \setminus \{(i,k)\}) \cup \{s,i\}$ is such that $|C'_e| = |C_e|$. The proof follows from the repeated application of this transformation. ■

Lemma 5 establishes the unique identifiability of vertices via edges in given minimal edge cuts. Hence a minimal X-j vertex cut C_v can be derived from a given edge cut C_e as follows:

$$\begin{aligned} &C_v = \varnothing \\ &\forall (i,k) \in C_e \\ &\qquad C_v := C_v \cup \begin{cases} i & \text{if } i \in Z \\ k & \text{if } k \in X \end{cases} \quad . \end{aligned}$$

Consequently, the minimal indegrees $|\underline{P}_j|$ of all intermediate vertices $j \in Z$ can be computed at $O(|E| \cdot |V|^2)$.

For the graph in Figure 5 (b) we find, for example, $C_e = \{(-1,1),(s,0)\}$. According to Lemma 5 the set $C'_e = \{(s,-1),(s,0)\}$ is a minimal edge cut too. Consequently, we get $C_v = \{-1,0\}$.

Suppose that j is the first vertex according to the topological order of all vertices with respect to dependence of the corresponding variables in Equation (1) whose current indegree $|P_j|$ is not minimal. The indegrees of all vertices k on paths that lead into j are minimal.

Lemma 6. *Consider $\dot{G}_j$ as in Equation (6) and with the vertices partitioned as in Equation (7). Paths from any $k \in Z_j \cup (P_j \setminus X_j)$ to j are unique.*

Proof. $|S_k| = 1 \; \forall k \in Z_j \cup (P_j \setminus X_j)$ as G is an SEU-graph. Hence, all these k have a single outgoing path leading into j. ■

Lemma 7. *Consider $\dot{G}_j$ as in Equation (6). Let the vertices be partitioned as in Equation (7). Paths from any $k \in Z_j$ to $j' \in P_j$ are unique if they exist.*

Proof. The proof follows immediately from that of Lemma 6. ■

Lemma 8. *There is always a minimal X-j vertex cut in $\dot{G}_j$ that contains only vertices from X_j and P_j.*

Proof. Let C_v be a minimal X-j vertex cut such that $C_v \not\subseteq X_j \cup P_j$. Let $k \in C_v$ and $k \notin X_j \cup P_j$. According to Lemma 7 the vertex k can be replaced by the unique $j' \in P_j$ such that the resulting vertex set C'_v is still a vertex cut with a cardinality that is less than or equal to that of C_v. Since C_v is minimal so is C'_v. ■

The key consequence of Lemma 8 is that a minimal X-j vertex cut in $\dot{G}_j$ is equivalent to one in the augmented bipartite graph $\dot{B}_j$ defined as

$$
\begin{aligned}
\dot{B}_j &= (\dot{V}_j^b, \dot{E}_j^b) \\
\dot{V}_j^b &= (s, X_j \cup P_j \cup \tilde{P}_j, j) \\
\dot{E}_j^b &= \{(i,k) | i,k \in \dot{V}_j^b \wedge (i \prec^* k \vee k \mathrel{\hat{=}} i)\} \cup \{(k,j) | k \in \tilde{P}_j\} \quad .
\end{aligned}
$$

$\tilde{P}_j$ contains a vertex k for each edge (i,j) where $i \in P_j \cap X_j$. We write $k \mathrel{\hat{=}} i$. An example is discussed in Section 4.

A set $C_v \subseteq X_j \cup P_j$ is an X-j vertex cut if and only if all edges in the bipartite subgraph $B_j = (V_j^b, E_j^b)$ of $\dot{B}_j$ that is obtained by removing s and j together with all their incident edges are incident with at least one vertex in C_v. Note that this property corresponds exactly to the definition of a *minimal vertex cover* in the bipartite graph B_j. According to Königs matching theorem [14] a minimal vertex cover is equivalent to a maximal matching in bipartite graphs. A maximal matching and a minimal vertex cover can be computed simultaneously by the well-known $O(\sqrt{|V_j^b|}|E_j^b|)$ algorithm of Hopcroft and Karp [13]. Its computational complexity undercuts that of the Ford-Fulkerson method for computing $|\underline{P}_j|$. Note that a minimal vertex cover is not necessarily unique. In the following we shall see that this fact has no negative impact on correctness and optimality of the proposed algorithm.

Theorem 2. *Consider an SEU-graph G. The following vertex elimination sequence yields the minimal number of multiplications and additions:*

[1] **FOR** $j = 1, \ldots, p$
[2] **IF** $|\underline{P}_j| < |P_j|$
[3] *reverse eliminate* $\{k \in Z_j \cup P_j | \nexists j' \in P_j : k \prec^* j' \wedge j' \in \underline{P}_j\}$
[4] *reverse eliminate* $\{k \in Z\}$.

Proof. We visit all vertices in the order of their indexes (line 1). Consider the first vertex $j \in Z$ where $|\underline{P}_j| < |P_j|$ (line 2). $\underline{P}_j$ is computed using the algorithm of Hopcroft and Karp.

To prove line 3 of the algorithm let $j' \in P_j$ such that $j' \notin \underline{P}_j$. Then $i \in \underline{P}_j \;\; \forall i \in X_j : i \prec^* j'$ as otherwise $\underline{P}_j$ is not an X-j vertex cut. The same applies to all $j'' \in Z_j$, $j'' \prec^* j'$ as paths emanating from j'' are unique according to Lemma 6. The elimination of all $j'' \in Z_j \cup (P_j \setminus X_j)$ with $j'' \prec^* j'$ transforms $\dot{G}_j$ such that only vertices in $\underline{P}_j$ are predecessors of j. Hence, the indegree of j becomes minimal no matter which minimal vertex cut $\underline{P}_j$ is used. Moreover, the reverse elimination of all $j'' \in Z_j \cup (P_j \setminus X_j)$ happens at minimal cost according to Lemma 1. Their outdegrees remain constant (equal to one), and their indegrees have been minimized during previous steps of the algorithm. Moreover, the k addressed in line 3 cannot be in X_j as $k \in \underline{P}_j$ if $k \in X_j \cap P_j$ and because $\prec^*$ is reflexive.

Following the forward part of the algorithm (lines 1–3) we reverse eliminate all remaining intermediate vertices. Again, their outdegrees remain unchanged (equal to one) while their indegrees have been minimized during the forward part of the algorithm. Consequently, all intermediate vertices are eliminated at the minimal possible cost in terms of the number of multiplications established by Lemma 1. The number of additions is constant according to Lemma 4. ■

Example. For the graph in Figure 3 the algorithm behaves as follows: Vertex 1 is visited first and $|\underline{P}_1| = |P_1| = 1$. For vertex 2 we get $1 = |\underline{P}_2| < |P_2| = 2$. We find $\{-1\}$ to be the corresponding minimal X-2 cut. Vertex 1 gets eliminated at the minimal cost of a single multiplication. This completes the forward part of the algorithm (lines 1–3).

The reverse part (line 4) is reduced to the elimination of the only remaining vertex, namely 2, at the cost of one multiplication. Hence, the graph is transformed into bipartite form at the overall minimal cost of two multiplications and a single addition. The latter is performed by any elimination sequence as shown in Lemma 4. □

4 Application and Further Test Results

A special case of Theorem 2 finds practical application in a research prototype of the differentiation-enabled NAGWare Fortran compiler [18, 19]. Typical representatives for SEU-graphs result from single scalar assignments in Fortran. The performance of tangent-linear codes generated by the forward mode of AD can be improved by preaccumulating the local gradients of all assignments followed by using these partial derivatives for the computation of the directional derivatives. The AD tool ADIFOR [3] implements statement-level reverse mode that is essentially equivalent to reverse vertex elimination in the corresponding computational graph. Obviously, this approach does not necessarily minimize the number of operations required locally. For example, the gradient of Equation (4) can be accumulated in forward vertex elimination mode at the cost of two multiplications (plus one addition) instead of the three multiplications (plus one addition) required by the reverse vertex elimination sequence. However, the limited size of typical right-hand sides of assignments in numerical codes suggests that this local application is unlikely to lead to significant improvements of the overall runtime of the tangent-linear program. The savings result primarily from the fact that the number of factors involved in the propagation of directional derivatives is decreased as the result of preaccumulation. The impact of the preaccumulation method is only secondary.

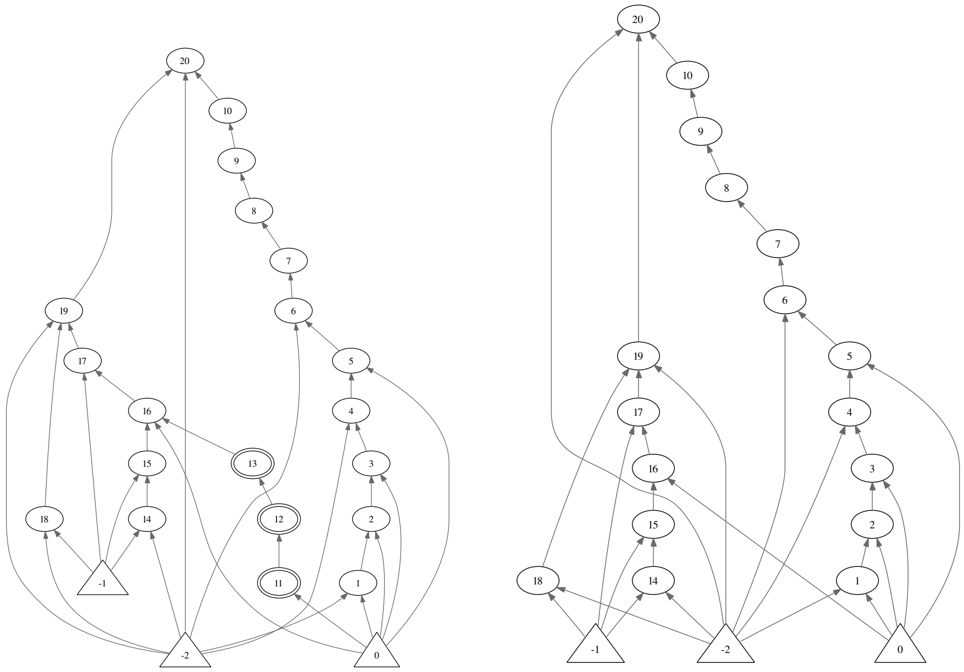
In the NAGWare Fortran compiler we assume that all elemental functions are at most binary, that is, $|P_j| \leq 2$ for all $j \in Z$. This observation leads to the following simplified version of the algorithm proposed in Theorem 2.

Lemma 9. *Consider an SEU-graph G with $|P_j| \leq 2$ for all $j \in Z$. The following vertex elimination sequence yields the minimal number of multiplications and additions:*

[1] **FOR** $j = 1, \ldots, p$
[2] **IF** $|P_j| = 1$

[3] *eliminate* j
[4] **FOR** $j = p, \ldots, 1$
[5] **IF** $j \in G$
[6] *eliminate* j

Proof. The optimality of the algorithm follows from Theorem 2. The elimination of all $j \in Z$ with $|P_j| = |S_j| = 1$ is performed at minimal cost. The remainder of the proof is lead by deriving a contradiction to the fact that all vertices j with $|P_j| = |S_j| = 1$ have been eliminated. Suppose that among the remaining vertex there is one with $|P_j| > |\underline{P}_j|$, that is $|P_j| = 2$ and $|\underline{P}_j| = 1$. If $\underline{P}_j = \{i\}$, then $i \in X$ by Lemma 6. It follows that there must be at least one k with $i \prec^* k \prec^* j$ such that $|P_k| = 1$. ■



(a) (b)

Fig. 6. Random SEU-Graph

The algorithm in Theorem 2 has been applied to a large number of random SEU-graphs. We have implemented both the graph generator and the elimination algorithm. The test results support the theoretical claims in this paper.

Consider, for example, the random SEU-graph in Figure 6 (a). The algorithm finds 16 to be the first (and only) vertex with nonminimal indegree. An auxiliary vertex $\tilde{0}$ is introduced to get the bipartite graph $B_{16} = (V_{16}^b, E_{16}^b)$ where $V_{16}^b = \{-2, -1, 0, 13, 15, \tilde{0}\}$ and $E_{16}^b = \{((-2, 15), (-1, 15), (0, 13), (0, \tilde{0})\}$. The vertex set $\{15, 0\}$ is found to be a minimal vertex cover in B_{16}. The vertices 13, 12, and 11 need to be reverse eliminated according to line 3 in Theorem 2. They are highlighted in Figure 6 (a). The elimination transforms the graph into that displayed in Figure 6 (b) at the cost of three multiplications and a single addition. It is easy to verify that all remaining vertices have minimal indegrees. Hence, reverse elimination makes the graph bipartite at an overall cost of 32 multiplications. Forward vertex elimination takes 38 multiplications whereas global reverse elimination takes 33 multiplications. The number of additions is 14 in all three cases.

5 Conclusion

The main contribution of this paper is a polynomial algorithm for the solution of a relevant subclass of the instances of an otherwise NP-complete problem. The savings that can be achieved by applying our algorithm as opposed to global reverse elimination in SEU-graphs is most likely bounded from above by a factor of two.[8] Hence, the computational speedup cannot be expected to be enormous. Moreover, the runtime of derivative code is probably dominated by memory access issues that have been ignored in this paper.

This paper reduces a relevant problem from the theory of automatic differentiation to classic graph algorithms. We believe that this novel insight contributes to an improved understanding of the computational complexity of derivative code. The results are likely to play an important role in future investigations in this area.

References

1. W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.
2. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
3. 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.
4. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2006. Springer.
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. SIAM, 1991.
7. J. Edmonds and R. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
8. L. Ford and D. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.
9. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
10. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[6]*, pages 126–135, 1991.
11. A. Griewank and O. Vogel. Analysis and exploitation of Jacobian scarcity. In *Proceedings of HPSC Hanoi.* Springer, 2003.
12. F. Harary. *Graph Theory.* Addison-Wesley, 1969.
13. J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comp.*, 2:225–231, 1973.
14. D. König. Graphs and matrices (hungarian). *Mat. Fiz. Lapok*, 38:116–119, 1931.

[8] We believe that the worst case for reverse vertex elimination is represented by $y = \sin(x) \cdot x \cdot \ldots x$ for the number of product evaluations going to ∞.

15. U. Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University Dresden, Dresden, Germany, Feb. 1999.
16. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004.
17. U. Naumann. The complexity of derivative computation. Technical Report AIB-2005-15, RWTH Aachen, August 2005.
18. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
19. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In *[4]*. 2006.
20. J. Utke. Flattening basic blocks. In *[4]*. 2006.