Edge Pushing is Equivalent to Vertex Elimination for Computing Hessians

Mu Wang* Alex Pothen† Paul Hovland‡

Abstract

We prove the equivalence of two different Hessian evaluation algorithms in AD. The first is the Edge Pushing algorithm of Gower and Mello, which may be viewed as a second order Reverse mode algorithm for computing the Hessian. In earlier work, we have derived the Edge Pushing algorithm by exploiting a Reverse mode invariant based on the concept of live variables in compiler theory. The second algorithm is based on eliminating vertices in a computational graph of the gradient, in which intermediate variables are successively eliminated from the graph, and the weights of the edges are updated suitably. We prove that if the vertices are eliminated in a reverse topological order while preserving symmetry in the computational graph of the gradient, then the Vertex Elimination algorithm and the Edge Pushing algorithm perform identical computations. In this sense, the two algorithms are equivalent. This insight that unifies two seemingly disparate approaches to Hessian computations could lead to improved algorithms and implementations for computing Hessians.

1 Introduction

Gower and Mello [1] have recently proposed an Edge Pushing algorithm to compute Hessians, which may be viewed as an algorithm that implements the second order Reverse mode in Algorithmic Differentiation. Wang, Gebremedhin and Pothen [2] have revisited this algorithm and offered a simpler derivation based on the notion of live variables from data flow analysis. Here, we offer a third interpretation of the new algorithm, as a symmetry-preserving vertex-elimination process on a gradient graph to compute the Hessian.We show that a major benefit of the Edge Pushing algorithm is that unlike a traditional vertex-elimination approach, it does not require the full gradient graph to be explicitly formed. With this new interpretation, we open the door to considering other elimination orderings in the vertex-elimination approach without explicitly forming the gradient graph.

*Department of Computer Science, Purdue University, West Lafayette IN 47907 USA (wangmu0701@gmail.com).

†Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (apothen@purdue.edu).

‡MCS Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne IL 60439 (hovland@mcs.anl.gov).

The Live Variables Algorithm for Hessians (LivarH) has been implemented paying careful attention to issues of efficiency and correctness, and statement-level preaccumulation has been incorporated into it to further improve performance [2]. The time complexity of the algorithm is bounded by $O(l \cdot s)$, where l is the number of elementary functions in the code list of the function, and s is the maximum number of live variables during the objective function evaluation. (Live variables will be defined in Section 4). The new algorithm can speed up the currently used compression-based methods that employ graph coloring for computing Hessians by factors of ten or more on a collection of test cases [2]. It also requires memory sizes smaller by a factor of two or three relative to the latter algorithms. The compression-based methods evaluate a Hessian-vector product in a single run, and the entire Hessian is recovered by combining the results of multiple Hessian-vector products. Hence these methods lack the ability to fully exploit the symmetry in the Hessian matrix. The new algorithm, in contrast, is able to exploit the symmetry and the sparsity in the Hessian during its evaluation, and this is a major advantage of the new approach.

2 Background on AD

2.1 Notations and Concepts of AD Given an objective function $\mathbf{y} = f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$ implemented as a program, the execution of the objective function can be decomposed as:

$$\begin{aligned} &\text{for } k = 1, \cdots, l: \\ &\quad v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}. \end{aligned}$$

See Fig 1 (a) for an example. In the decomposition, each $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$ represents an elemental function (we will also call it a single assignment code (SAC)). In each step, the value of v_k results from evaluating the elemental function φ_k, which takes all the variables $\{v_i : v_i \prec v_k\}$ as operands. Here $v_i \prec v_k$ denotes that variable v_k directly *depends* on variable v_i. We read v_i precedes v_k, or v_i is a predecessor of v_k. We can also write $v_k \succ v_i$ to denote the same relationship, and read v_k succeeds v_i, or that v_k is a successor of v_i. The set $\{v_i : v_i \prec v_k\}$ represents all precedents of v_k. Following the convention in [3], we assume $\{v_{1-n}, \cdots, v_0\}$ are the

n independent variables $\mathbf{x}$, $\{v_{l-m+1}, \cdots, v_l\}$ are the m dependent variables $\mathbf{y}$, and the remaining variables v_j, where $1 \leq j \leq l-m$, are intermediate variables. Since we are interested in evaluating Hessian matrices in this paper, we assume that the objective function f is a scalar function, and hence $m = 1$. Also, l denotes the total number of SACs, which represents the complexity of the objective function. The objective function f can be computed by evaluating the SAC sequence.

The SAC sequence defines a computational graph $G = (V, E)$ of the objective function. The vertices V are the independent variables, dependent variables and intermediate variables. The edges E represent dependency relations between variables, i.e., an edge $(v_i, v_k) \in E$ if and only if $v_i \prec v_k$. See Fig 1(b) for an example. In the graph G, a variable is a dependent variable if and only if it has out-degree zero, a variable is an independent variable if and only if it has in-degree zero, and all other variables are intermediate variables $(v_1, \cdots, v_{l-m})$. The computational graph contains all the information about the SAC sequence. Since most elemental functions are unary or binary, the computational graph often has the property that the in-degree of a vertex is at most two. In the computational graph, we define v_k as a *common successor* of v_i and v_j if $v_i \prec v_k$ and $v_j \prec v_k$. In this case v_i is a *neighbor* of v_j, and vice versa.

2.2 Vertex Elimination For Jacobian Given a computational graph $G = (V, E)$, we can augment each edge (v_i, v_k) with a weight $c(i, k)$ that represents the local partial derivative $c(i, k) = \frac{\partial v_k}{\partial v_i}$, when $v_i \prec v_k$. (We can assume all other edges $c(i, k)$ have weight 0 when $v_i \not\prec v_k$, and we use the notation $c(i, k)$ instead of $c(v_i, v_k)$ for simplicity.) Then the Jacobian of the objective function can be computed via a *vertex elimination* procedure, studied by Griewank and Reese and Naumann, and described in Algorithm 1 [4, 5]. Fig. 1 illustrates this process for an example function and its computational graph.

The vertex elimination procedure has several properties stated as follows.

- The rule for eliminating a vertex v_j is to add an edge (v_i, v_k), for all $v_i \prec v_j$ and $v_j \prec v_k$, with weight $c(i, j) \cdot c(j, k)$. If the edge (v_i, v_k) already exists, we add this weight to the existing edge (v_i, v_k). So the number of updates of edge weights for eliminating a vertex v_j is its current Markowitz degree, the product of its indegree and its outdegree.

- The order in which vertices are eliminated does not affect the final result, and we can eliminate the intermediate variables in any order. Figure 1 (c) and (d) gives an example where we first eliminate v_2 and then v_1 on the computational graph in Figure 1 (b). Eliminating v_1 and then v_2 will give the same result. However, the number of operations and the intermediate storage needed to compute the Jacobian will depend on the elimination ordering. Finding an optimal elimination order that minimizes the number of operations is an NP-complete problem [6].

- Now we consider the closure of the $\prec$ relation, $\prec^*$, which relates a vertex to all of its ancestors in the computational graph (i.e., vertices related to it by a sequence of $\prec$ relations). A path in the computational graph joins each ancestor to a vertex v_i, and we define the weight of this path as the product of all the weights of the edges on this path. Then we have

$$\frac{\partial v_k}{\partial v_i} = \sum_{P|\text{path from } i \text{ to } k} w(P).$$

  The correctness of the elimination algorithm exploits this fact.

Algorithm 1 Vertex elimination for computing the Jacobian

Input: The computational graph $G = (V, E)$ augmented with local partial derivatives $c(i, k) = \frac{\partial v_k}{\partial v_i}$ as edge weights

```
while G = (V, E) still contains intermediate vertex do
    Pick an intermediate vertex v_j
    for all v_i : v_i ≺ v_j do
        for all v_k : v_j ≺ v_k do
            c(i, k) += c(i, j) · c(j, k)
        end for
    end for
    Remove v_j from G
end while
```

Output: The remaining edges in G represent the Jacobian of the objective function f.

2.3 First Order Non-Incremental Reverse Mode Though the Vertex Elimination algorithm on the computational graph can compute the gradient of the objective function, there are some difficulties in practically implementing this algorithm. The main issue is that to eliminate a vertex v_j, we need to know all predecessors and successors of v_j, which requires the computational graph to be explicitly generated from the

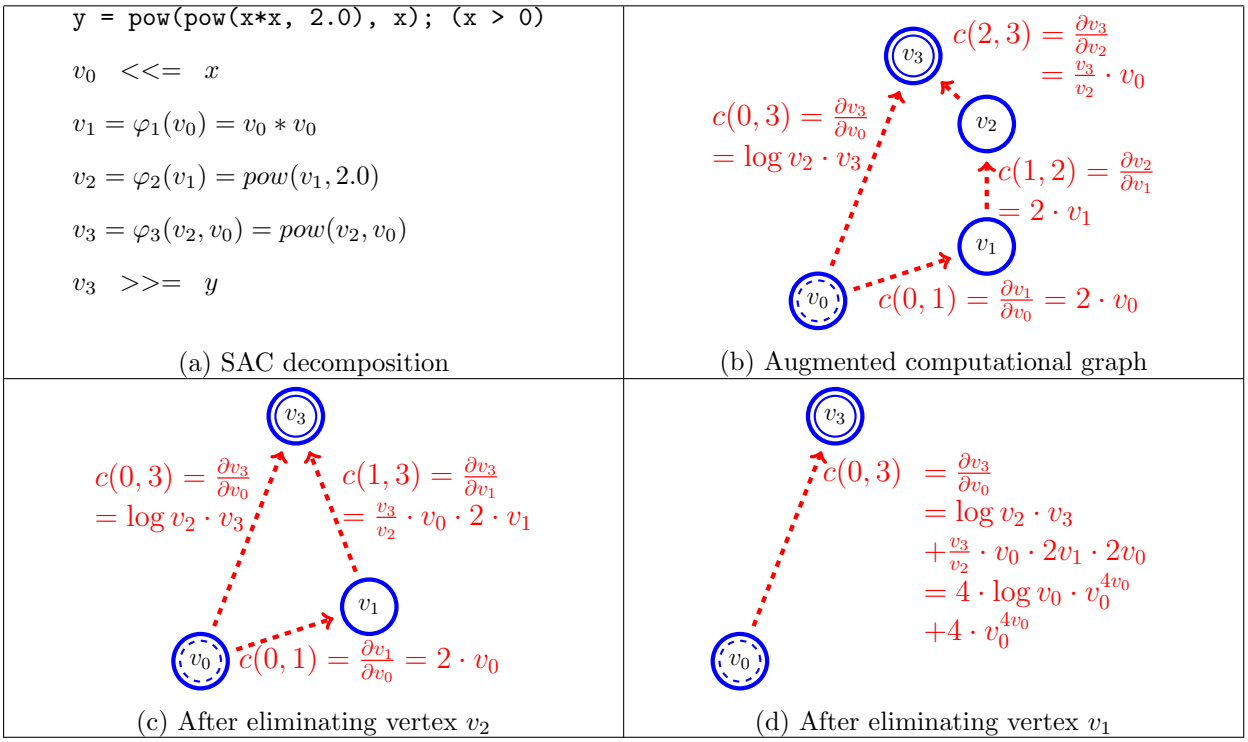


Figure 1: SAC decomposition and an illustration of vertex elimination on the corresponding computational graph.

SAC sequence.

Two major practical modes of AD which evaluate the first order derivatives are known as the Forward mode and the Reverse mode. They can be viewed as two extreme orderings for applying the chain rule on composite functions defined by the SAC sequence. The forward mode evaluates the derivatives in the same order in which the SAC sequence is evaluated. The reverse mode evaluates the derivatives in an order *opposite* to one in which the SAC sequence is evaluated. The reverse mode has a time complexity proportional to the number of dependent variables and the complexity of the objective function [7]. Hence for a scalar function, the complexity of evaluating the gradient using reverse mode is a constant times the complexity of the objective function. The cost is that reverse mode requires a trace of the SAC sequence (the values of the intermediate variables needed to evaluate the derivatives) to be stored so that a reverse sweep on the SAC sequence is possible.

There are two further variations of the first order reverse mode: incremental and non-incremental versions. They are mathematically equivalent but differ in the order in which they compute the derivative values.

Algorithm 2 describes the first order incremental reverse mode. The elemental derivatives $\overline{v}_i \equiv \frac{\partial f}{\partial v_i}$ are called the adjoints. The SACs $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$ are processed in the order $k = l, l-1, \cdots, 1$ (reverse order). In each step when processing $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$, the algorithm updates the adjoints of all predecessors of v_k, i.e, $\overline{v}_i$ where $v_i \prec v_k$. The output of the algorithm is the gradient of the objective function, $\overline{v}_i = \frac{\partial f}{\partial v_i}, 1 - n \leq i \leq 0$. The algorithm is the incremental version because each update just adds one term to a partially computed value of the adjoint. The incremental version is practically preferred because all predecessors of v_k can be obtained using only local information about the SAC sequence (it is directly available in the processed SAC $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$); hence it can be efficiently implemented.

Algorithm 2 First order incremental reverse mode

Input: The objective function f as a SAC sequence $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$

1: $\overline{v}_l = 1, \overline{v}_{l-1} = \cdots = \overline{v}_{1-n} = 0$
2: **for** $k = l-1, l-2, \cdots, 1$ **do**
3: **for all** $v_i \prec v_k$ **do**
4: $\overline{v}_i += \frac{\partial v_k}{\partial v_i} \overline{v}_k$
5: **end for**
6: **end for**

Output: The gradient $\nabla f = \{\overline{v}_{1-n}, \cdots, \overline{v}_0\}$

Algorithm 3 First order non-incremental reverse mode

Input: The objective function f as a SAC sequence $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$

1: $\overline{v}_l = 1$
2: **for** $i = l-1, l-2, \cdots, 1-n$ **do**
3: $\quad \overline{v}_i = \sum_{v_k \succ v_i} \frac{\partial v_k}{\partial v_i} \overline{v}_k$
4: **end for**

Output: The gradient $\nabla f = \{\overline{v}_{1-n}, \cdots, \overline{v}_0\}$

For the first order nonincremental reverse mode shown in Algorithm 3, in each step, the adjoint $\overline{v}_i$ is determined by summing the contributions from all its successors $v_k, v_i \prec v_k$. So in each step we need to know all successors of v_i. A disadvantage of the nonincremental mode is that it requires global information about the SAC sequence to determine all successors of v_i. However, the nonincremental reverse mode is theoretically important because it can be used to generate a computational graph of the gradient as we discuss in the next Section.

3 Evaluating Hessian via Vertex Elimination

3.1 Computational Graph of Gradient We begin by augmenting the computational graph of a scalar objective function into a computational graph of its gradient. The construction views the assignment of each adjoint value $\overline{v}_i$ in the first order non-incremental reverse mode as an "elemental" function, i.e, a single assignment code for $\overline{v}_i$. Thus $\overline{v}_i = \sum_{v_k \succ v_i} \frac{\partial v_k}{\partial v_i} \overline{v}_k$ is viewed as a function of all $\overline{v}_k$ where v_k is a successor of v_i and all v_j where v_j is a neighbor of v_i.

Then we append the non-incremental reverse mode for computing the gradient after the SAC sequence of the objective function, as shown in Figure 3. Hence symbolically we have an SAC sequence which evaluates the objective function and its gradient, and correspondingly we can associate a computational graph with it, that we will denote the computational graph of the gradient, G_g[1].

Figure 2(a) gives an example of the computational graph of the gradient for the function given by Figure 1(a). The vertices of the graph G_g consist of two disjoint yet symmetric sets V and $\overline{V}$. The vertex set V represents the variables involved in the function evaluation $v_{1-n}, \cdots, v_l$, and the vertex set $\overline{V}$ represents the adjoints for each variable in the non-incremental reverse mode $\overline{v}_{1-n}, \cdots, \overline{v}_l$. The vertices v_i in V are also called primal vertices and the vertices $\overline{v}_i$ in $\overline{V}$ are also called dual/adjoint vertices. In the graph G_g, $v_{1-n}, \cdots, v_0$ are independent variables, and $\overline{v}_{1-n}, \cdots, \overline{v}_0$ are dependent variables.

The edges of G_g consist of three parts : E_G, $E_{\overline{G}}$ and E_C. We use different colors to represent different kinds of edges in Fig. 2(a). The set E_G is colored red, $E_{\overline{G}}$ is colored violet, and E_C is colored green. The edge weights are also annotated accordingly.

The set E_G consists of edges of the form $(v_i, v_k) \in E_G \iff v_i \prec v_k$, with weight $c(i,k) = \frac{\partial v_k}{\partial v_i}$. In fact, (V, E_G) is the computational graph of the objective function embedded in the computational graph of the gradient. The set $E_{\overline{G}}$ consists of edges of the form $(\overline{v}_k, \overline{v}_i) \in E_{\overline{G}} \iff \overline{v}_k \prec \overline{v}_i \iff v_i \prec v_k$. The equivalence can be seen by taking partial derivatives of $\overline{v}_i$ w.r.t $\overline{v}_k$, which yields

$$c(\overline{k}, \overline{i}) = \frac{\partial}{\partial \overline{v}_k}[\overline{v}_i] = \frac{\partial}{\partial \overline{v}_k}[\sum_{v_j \succ v_i} \frac{\partial v_j}{\partial v_i} \overline{v}_j] = \frac{\partial v_k}{\partial v_i}.$$

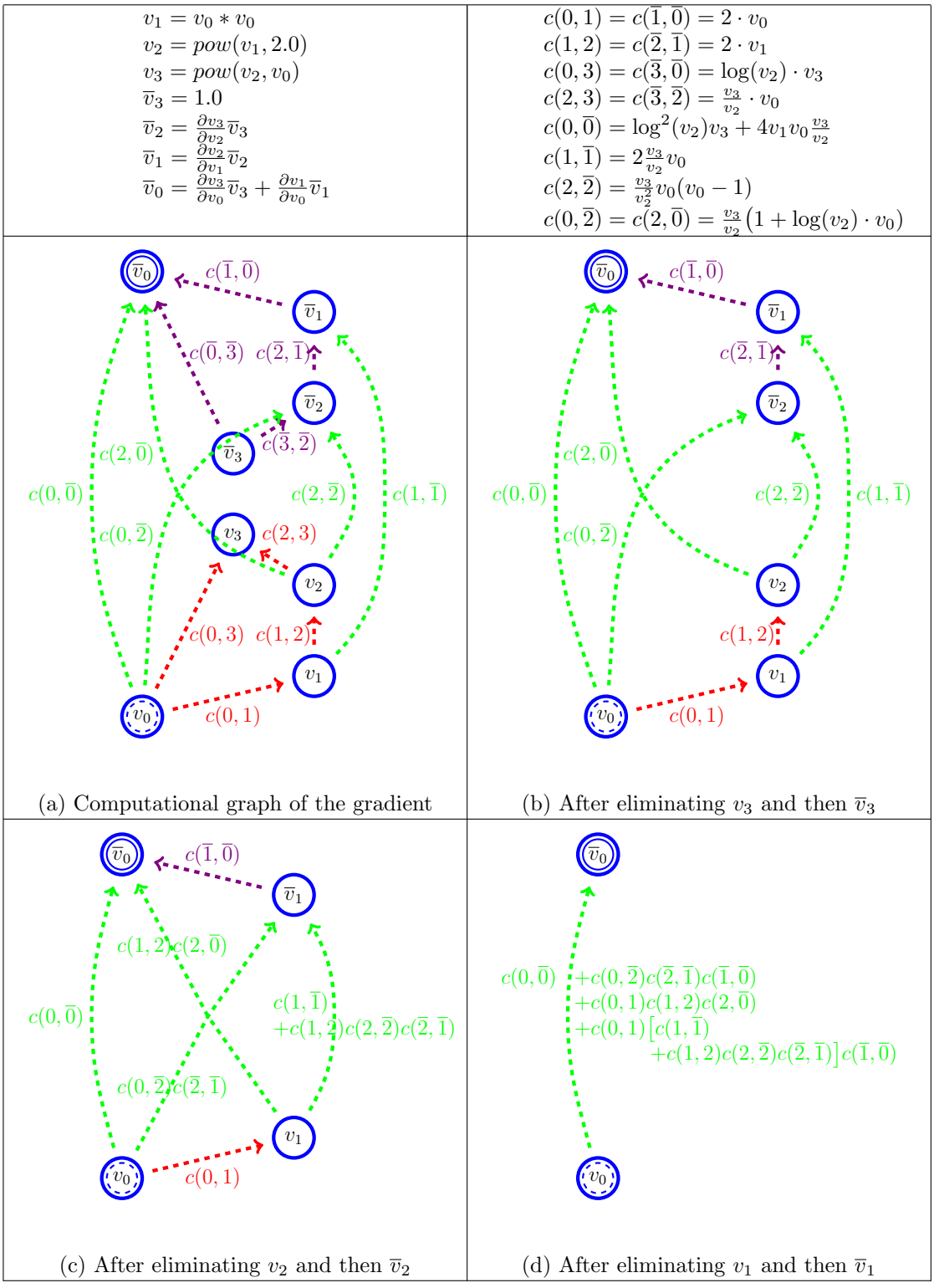
We can think of $(\overline{V}, E_{\overline{G}})$ as a symmetric image of (V, E_G) with the edge directions reversed and the edge weights preserved. Finally the set E_C consists of edges of the form $(v_i, \overline{v}_j) \in E_C \iff \exists v_k, s.t,\ v_i \prec v_k, v_j \prec v_k$, with $c(i, \overline{j}) = \sum_{v_i \prec v_k, v_j \prec v_k} \frac{\partial^2 v_k}{\partial v_i \partial v_j} \overline{v}_k$. These edges arise when we compute a partial derivative of $\overline{v}_j$ w.r.t v_i, which yields

$$\begin{aligned} c(i, \overline{j}) &= \frac{\partial}{\partial v_i}[\overline{v}_j] = \frac{\partial}{\partial v_i}[\sum_{v_k \succ v_j} \frac{\partial v_k}{\partial v_j} \overline{v}_k] \\ &= \sum_{v_i \prec v_k, v_j \prec v_k} \frac{\partial^2 v_k}{\partial v_i \partial v_j} \overline{v}_k. \end{aligned} \tag{3.1}$$

This part is symmetric, i.e., $c(i, \overline{j}) = c(j, \overline{i})$. Notice that v_k is a common successor of v_i and v_j. Also there is no ordering relation between v_i and v_j, and it is possible to have $v_i = v_j$.

3.2 Vertex Elimination for Hessian If we run the Vertex Elimination algorithm on the graph G_g, the result will be the first order derivative of the gradient, which is the Hessian of the original objective function, $H_{ij} = \frac{\partial \overline{v}_j}{\partial v_i} = \frac{\partial^2 f}{\partial v_i \partial v_j}$. Figure 2(b-d) gives an example of one possible vertex elimination sequence on Figure 2(a). The elimination order is $\{v_3, \overline{v}_3, v_2, \overline{v}_2, v_1, \overline{v}_1\}$. Notice that since vertex elimination is order independent, any other elimination order will also give the same

[1] In [3], the graph G_g is called Hessian graph since the Hessian can be computed using it. Here we use the term *computational graph of the gradient* to be consistent with the definition of the computational graph G of the objective function. First order algorithms working on G give the first order derivatives of the objective function, i.e, the gradient. Similarly first order algorithms working on G_g give the first order derivatives of the gradient, i.e, the Hessian.



(a) Computational graph of the gradient

(b) After eliminating v_3 and then $\overline{v}_3$

(c) After eliminating v_2 and then $\overline{v}_2$

(d) After eliminating v_1 and then $\overline{v}_1$

Figure 2: Vertex elimination on the computational graph of the gradient of Figure 1(a). From (a) to (b), vertices v_3 and $\overline{v}_3$ are eliminated. As there are no outgoing edges for v_3 and no incoming edges for $\overline{v}_3$, no edge will be created. From (b) to (c), vertices v_2 and $\overline{v}_2$ are eliminated. When eliminating v_2, an edge $c(1,\overline{2}) = c(1,2)c(2,\overline{2})$ is created. Then when eliminating $\overline{v}_2$, we add $c(1,2)c(2,\overline{2})c(\overline{2},\overline{1})$ to $c(1,\overline{1})$. From (c) to (d), vertices v_1 and $\overline{v}_1$ are eliminated. The final result is the second order derivative of f.

Input: Initial values of $\{v_{1-n}, \cdots, v_0\}$
For $k = 1, \cdots, l$
$v_k = \varphi_k(v_j)_{\{v_j : v_j \prec v_k\}}$
$\overline{v}_l = 1$
For $i = l-1, \cdots, 1-n$
$\overline{v}_i = \sum_{v_k \succ v_i} \frac{\partial v_k}{\partial v_i} \overline{v}_k$
Output: $\{\overline{v}_{1-n}, \cdots, \overline{v}_0\}$

Figure 3: Augmented objective function f_G

final Hessian, although the number of operations and the intermediate storage needed could differ with the ordering. Griewank and Walther have conjectured [3] (Chapter 10) that the optimal elimination order on G_g should preserve symmetry, i.e, a primal and its dual vertex $(v_i, \overline{v}_i)$ should be eliminated one after another. Some empirical evidence for the conjecture can be found in Table 1 of [8].

As discussed earlier, to perform vertex elimination on the computational graph of the gradient G_g, we require this graph to be explicitly constructed. To do this, we need to know not only all predecessors and successors of every vertex v_i, but also all *neighbors* of every vertex v_i. The need to know global information on the SAC sequence impacts the efficiency of the algorithm.

4 Evaluating Hessian via Edge Pushing

Algorithm 4 Non-incremental second order reverse mode (Edge Pushing) algorithm

Input: The objective function f as a SAC sequence $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$

1: $S_{l+1} = \{v_l\}$, $h_{l+1}(v_l, v_l) = 0$, $\overline{v}_l = 1$, $\overline{v}_{l-1} = \cdots = \overline{v}_{1-n} = 0$
2: **for** $k = l, \cdots, 1$ **do**
3: $\quad S_k = S_{k+1} \setminus \{v_k\} \cup \{v_i : v_i \prec v_k\}$
4: $\quad$ **for all** $v_i \prec v_k$ **do**
5: $\quad\quad \overline{v}_i = \overline{v}_i + \frac{\partial v_k}{\partial v_i} \overline{v}_k$
6: $\quad$ **end for**
7: $\quad$ **for all** unordered pairs (v_i, v_j) over S_k **do**
8: $\quad\quad h_k(v_i, v_j) = h_{k+1}(v_i, v_j) + \frac{\partial v_k}{\partial v_i} h_{k+1}(v_j, v_k) + \frac{\partial v_k}{\partial v_j} h_{k+1}(v_i, v_k) + \frac{\partial v_k}{\partial v_i} \frac{\partial v_k}{\partial v_j} h_{k+1}(v_k, v_k) + \frac{\partial^2 v_k}{\partial v_j \partial v_k} \overline{v}_k$
9: $\quad$ **end for**
10: **end for**

Output: Hessian for f as $\frac{\partial^2 f}{\partial v_i \partial v_j} = h_1(v_i, v_j)$

The *Edge Pushing* (EP) algorithm of Gower and Mello [1], may be viewed as a second order reverse mode algorithm for the Hessian. In earlier work, we have provided a simpler derivation of the algorithm based on a data flow analysis (from compiler theory) of the reverse mode of AD [2]. During the function evaluation, a variable is *live* if it holds a value that *will be* used in the future. In the reverse mode for computing the derivatives of a scalar function, referring to the first-order incremental reverse mode, Algorithm 2, the initial live variable set consists of the dependent variable v_l, and the final live variable set is the set of independent variables. We denote the set of live variables at the end of step k by S_k. At the kth step, when an elemental function $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$ is processed, the set of live variables is updated as

$$S_k = S_{k+1} \cup \{v_i : v_i \prec v_k\} \setminus \{v_k\}.$$

We can also express the objective function as an equivalent function defined in terms of only the live variables at each step. Referring to Algorithm 2, initially $f_{l+1}(S_{l+1}) = v_l$. At the k-th step, we treat v_k in f_{k+1} as a composite function by replacing it by $v_k = \varphi_k(v_i : v_i \prec v_k)$, and thus obtain an equivalent function defined in terms of the current live variables $f_k(S_k)$. An invariant in the first order incremental reverse mode is that the adjoints computed in each step during the algorithm are exactly the first order derivatives of the dependent variable w.r.t current live variables in the step. Figure 4 shows the live variable sets S_k and the equivalent functions $f_k(S_k)$ for the example function in this paper.

The following proposition explicitly states how the invariant can be extended to second order.

PROPOSITION 4.1. *In reverse mode, after processing the SAC $v_k = \varphi_k(v_i)_{\{v_i : v_i \prec v_k\}}$, the intermediate results we maintain are the first and second order derivatives of the equivalent function $f_k(S_k)$.*

The relation between the derivatives of the functions $f_k(S_k)$ and $f_{k+1}(S_{k+1})$ is determined by the second order chain rule:

$$\begin{aligned}(4.2)\quad \frac{\partial^2 f_k(S_k)}{\partial v_i \partial v_j} &= \frac{\partial^2 f_{k+1}(S_{k+1})}{\partial v_i \partial v_j} + \frac{\partial v_k}{\partial v_i} \frac{\partial^2 f_{k+1}(S_{k+1})}{\partial v_j \partial v_k} \\ &+ \frac{\partial v_k}{\partial v_j} \frac{\partial^2 f_{k+1}(S_{k+1})}{\partial v_i \partial v_k} + \frac{\partial v_k}{\partial v_i} \frac{\partial v_k}{\partial v_j} \frac{\partial^2 f_{k+1}(S_{k+1})}{\partial^2 v_k} \\ &+ \frac{\partial^2 v_k}{\partial v_i \partial v_j} \frac{\partial f_{k+1}(S_{k+1})}{\partial v_k}.\end{aligned}$$

The Edge Pushing algorithm is a transcript of Eq (4.2), and is described in Algorithm 4. In the

algorithm, we use $h_k(v_i, v_j)$ to denote $\frac{\partial^2 f_k(S_k)}{\partial v_i \partial v_j}$. At termination, the set of live variables S_1 will be the set of independent variables, and h_1 will be the Hessian of the objective function. Figure 4 is an example of how Edge Pushing is performed on the objective function defined in Figure 1(a). The complexity of the algorithm can be shown to be $O(l \cdot s)$, where s is the maximum size of all live variable sets during the algorithm [2]. The complexity can be reduced by optimizing the algorithm to exploit sparsity, and the details may be found in [2].

5 Equivalence between Vertex Elimination and Edge Pushing

5.1 Parallel Edges on Computational Graph of Gradient Before we can build the connection between Vertex Elimination and Edge Pushing, we introduce a finer definition of the computational graph of the gradient. Previously we defined all edges in E_C to be $(v_i, \overline{v}_j) \in E_C \iff \exists v_k, s.t, v_i \prec v_k$ and $v_j \prec v_k$, and the weight $c(i,\overline{j}) = \sum_{v_i \prec v_k, v_j \prec v_k} \frac{\partial^2 v_k}{\partial v_i \partial v_j} \overline{v}_k$. The weight $c(i,\overline{j})$ is the summation of $\frac{\partial^2 v_k}{\partial v_i \partial v_j}\overline{v}_k$ over all common successors v_k of v_i and v_j.

Now we decompose each edge $(v_i, \overline{v}_j)$ into a set of parallel edges, where each edge $(v_i, \overline{v}_j)^{v_k}$ only carries the weight $c^k(i,\overline{j}) = \frac{\partial^2 v_k}{\partial v_i \partial v_j}\overline{v}_k$ of a single common successor v_k of v_i and v_j. For example, the edge $c(0,\overline{0})$ in Figure 2 becomes two edges $c^1(0,\overline{0})$ and $c^3(0,\overline{0})$. The summation of $c^k(i,\overline{j})$ over all common successors v_k will give the final value of $c(i,\overline{j})$. From now on we will only consider the edges in E_C in the form of $c^k(i,\overline{j})$. See Figure 5(a) for an example. Notice that only edges in E_C are considered as parallel edges, and that edges in E_G, $E_{\overline{G}}$ and E_H (to be defined shortly) are all single edges.

5.2 Edge Pushing as Vertex Elimination We will prove that the Edge Pushing algorithm does the same computations as vertex elimination on the graph G_g when vertices are eliminated in the order $\{v_l, \overline{v}_l, \cdots, v_k, \overline{v}_k, \cdots, v_1, \overline{v}_1\}$. As vertices are eliminated, we add a set of edges on G_g denoted E_H following two simple rules. The first rule is that newly added edges are added only to E_H. The second rule is that when eliminate v_k, we move all edges of the form $(v_i, \overline{v}_j)^{v_k}$ from E_C to E_H. See Figure 5(b, c, d) for an example, in which edges in E_H are drawn in black color. Then we can prove the following lemma for the Vertex Elimination algorithm on G_g with the elimination order specified above.

LEMMA 5.1. *E_H will only have edges of the type $c(i,\overline{j})$.*

Proof. We prove the result by induction. Initially E_H is empty, so the claim is true. To eliminate the primal vertex v_k, we do the following.

All edges in E_G adjacent to v_k are of the form (v_i, v_k), because we are eliminating vertices in a reverse topological order of G. The set of edges $E_{\overline{G}}$ has no edges incident on v_k (having it as an endpoint). The set E_C has no edges incident on v_k. Since we are following a reverse topological order of G, all successors v_u of v_k have been already eliminated. Hence every edge in the form of $(v_k, \overline{v}_i)^{v_u}$ originally in E_C has been already moved to E_H before we reach v_k. Combining these three observations, when eliminating a vertex v_k the newly added edges are only of the type $(v_i, \overline{v}_j)$, where $(v_i, v_k) \in E_G$ and $(v_k, \overline{v}_j) \in E_H$. When we move all edges $(v_i, \overline{v}_j)^{v_k}$ from E_C to E_H the claim continues to hold.

Similarly, assume the claim is true before the elimination of $\overline{v}_k$. Then to eliminate the dual vertex $\overline{v}_k$: The set E_G has no edge incident on $\overline{v}_k$. Edges in $E_{\overline{G}}$ incident on $\overline{v}_k$ are of the form $(\overline{v}_k, \overline{v}_j)$. The set E_C has no edge incident on $\overline{v}_k$. When eliminating $\overline{v}_k$ the newly added edges will only be of the type $(v_i, \overline{v}_j)$, where $(v_i, \overline{v}_k) \in E_H$, and $(\overline{v}_k, \overline{v}_j) \in E_{\overline{G}}$. This completes the proof.

LEMMA 5.2. *After eliminating v_k and then $\overline{v}_k$, E_H is the Hessian of the equivalent function $f_k(S_k)$.*

Proof. This is proved by showing that after eliminating the vertex pair $(v_k, \overline{v}_k)$, the edges in E_H exactly match all nonzero entries in $h_k(S_k, S_k)$. Initially, E_H is empty and we have $h_{l+1}(v_l, v_l) = 0$, which are in agreement.

Assume the claim is true before the elimination of v_k. Then consider the computation for eliminating v_k. The newly added edges will only be of the form $(v_i, \overline{v}_j)$, where $(v_i, v_k) \in E_G$ and $(v_k, \overline{v}_j) \in E_H$. This means the weights of edges in E_H are updated as follows.

$$\begin{aligned} c'(i,\overline{j}) &= c(i,\overline{j}) + c(i,k)\cdot c(k,\overline{j}) \\ &= c(i,\overline{j}) + \frac{\partial v_k}{\partial v_i}\cdot c(k,\overline{j}), \qquad j \neq k, \\ c'(i,\overline{k}) &= c(i,\overline{k}) + c(i,k)\cdot c(k,\overline{k}) \\ &= c(i,\overline{k}) + \frac{\partial v_k}{\partial v_i}\cdot c(k,\overline{k}). \end{aligned}$$

Next we move the edge $c^k_{i\overline{j}}$ from E_C to E_H. Hence we have

$$c''(i,\overline{j}) = c'(i,\overline{j}) + c^k(i,\overline{j}) = c'(i,\overline{j}) + \frac{\partial^2 v_k}{\partial v_i \partial v_j}\overline{v}_k.$$

The computation done for eliminating $\overline{v}_k$ creates new edges only of the form $(v_i, \overline{v}_j)$, where $(v_i, \overline{v}_k) \in E_H$ and $(\overline{v}_k, \overline{v}_j) \in E_{\overline{G}}$.

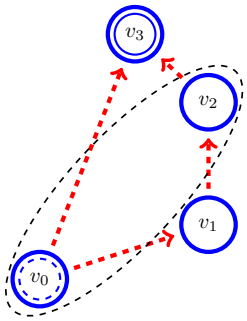
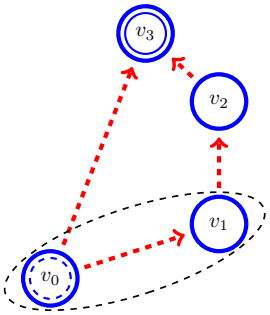
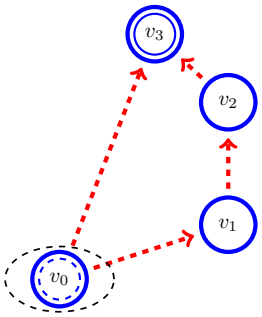
$k=$	3	2	1
SAC :	$v_3 = pow(v_2, v_0)$	$v_2 = pow(v_1, 2.0)$	$v_1 = v_0 * v_0$
S_k :	$\{v_0, v_2\}$	$\{v_0, v_1\}$	$\{v_0\}$
$f_k(S_k)$:	$f(v_0, v_2) = v_2^{v_0}$	$f(v_0, v_1) = v_1^{2v_0}$	$f(v_0) = (v_0)^{4v_0}$
$h_k(S_k, S_k)$:	$h_3(v_0, v_0) = v_2^{v_0}\log^2(v_2)$ $h_3(v_0, v_2) = v_0 v_2^{v_0-1}\log(v_2) + v_2^{v_0-1}$ $h_3(v_2, v_2) = v_0(v_0-1)v_2^{v_0-2}$	$h_2(v_0, v_0) = 4v_1^{2v_0}\log^2(v_1)$ $h_2(v_0, v_1) = 4v_0 v_1^{2v_0-1}\log(v_1) + 2v_1^{2v_0-1}$ $h_2(v_1, v_1) = 2v_0(2v_0-1)v_1^{2v_0-2}$	$h_1(v_0, v_0) = 4v_0^{4v_0-1} + 16v_0^{4v_0}\log(v_0)v_0^{4v_0} + 16\log^2(v_0)v_0^{4v_0}$
			

Figure 4: An illustration of the Edge Pushing algorithm. The live variable sets S_k, equivalent functions $f_k(S_k)$, and the Hessians of equivalent functions $h_k(S_k, S_k)$ are listed. The live variables are also shown in dashed ellipses.

$$\begin{aligned} c'''(i,\bar{j}) &= c''(i,\bar{j}) + c'(i,\bar{k})\cdot c(\bar{k},\bar{j}) \\ &= c''(i,\bar{j}) + \frac{\partial v_k}{\partial v_j}\cdot c'(i,\bar{k}). \end{aligned}$$

Combining these equations, we obtain

$$\begin{aligned} c'''(i,\bar{j}) &= c(i,\bar{j}) + \frac{\partial v_k}{\partial v_i}\cdot c(k,\bar{j}) \\ &+ \frac{\partial v_k}{\partial v_j}\cdot c(i,\bar{k}) + \frac{\partial v_k}{\partial v_i}\frac{\partial v_k}{\partial v_j}\cdot c(k,\bar{k}) + \frac{\partial^2 v_k}{\partial v_i \partial v_j}\overline{v}_k. \end{aligned}$$

Finally we compare the result with that from the Edge Pushing algorithm. For the latter, we have

$$\begin{aligned} h_k(v_i,v_j) &= h_{k+1}(v_i,v_j) + \frac{\partial v_k}{\partial v_i}h_{k+1}(v_j,v_k) \\ &+ \frac{\partial v_k}{\partial v_j}h_{k+1}(v_i,v_k) + \frac{\partial v_k}{\partial v_i}\frac{\partial v_k}{\partial v_j}h_{k+1}(v_k,v_k) + \frac{\partial^2 v_k}{\partial v_j \partial v_k}\overline{v}_k. \end{aligned}$$

Comparing the last two displayed equations term by term, we see that the computation in going from $c_{i\bar{j}}$ to $c'''_{i\bar{j}}$ in the Vertex Elimination algorithm is exactly the computation in going from h_{k+1} to h_k.

THEOREM 5.1. *The Vertex Elimination algorithm on the gradient graph G_g with a symmetry-preserving reverse topological ordering performs the same computation in each step as the Edge Pushing algorithm.*

5.3 Discussion We have proved that the Edge Pushing algorithm can be interpreted as a symmetry-preserving Vertex Elimination algorithm that eliminates the primal vertices in reverse topological order and the dual vertices in topological order. Then it is natural to ask if Edge Pushing is an optimal form of Vertex Elimination, that is, whether or not the number of updates on edge weights is minimized. Unfortunately, it is not.

We can construct a counter-example to show that the reverse topological order is not optimal when preserving symmetry. In the example, the graph G has a structure that each of its n independent variables passes through h unary functions, and the results are merged via a binary tree structure (binary functions). Figure 6 shows the case where $n = 4$ and $h = 3$. The optimal symmetry-preserving elimination order in this example should first eliminate the intermediate variables on the unary chain, and then eliminate vertices of the binary tree from leaf to root, which is the order given by the smallest Markowitz degree. In this order, only 3 edge updates are needed for eliminating each pair of vertices $(v_{ij}, \overline{v}_{ij})$ on the unary chain. The reverse topological order first eliminates the binary tree from root to leaf, and then eliminates the unary chain. In this order, $n(n+1)/2$ edge updates are needed to eliminate each pair of vertices on the unary chain.

As discussed in Section 3.2, it has been conjectured

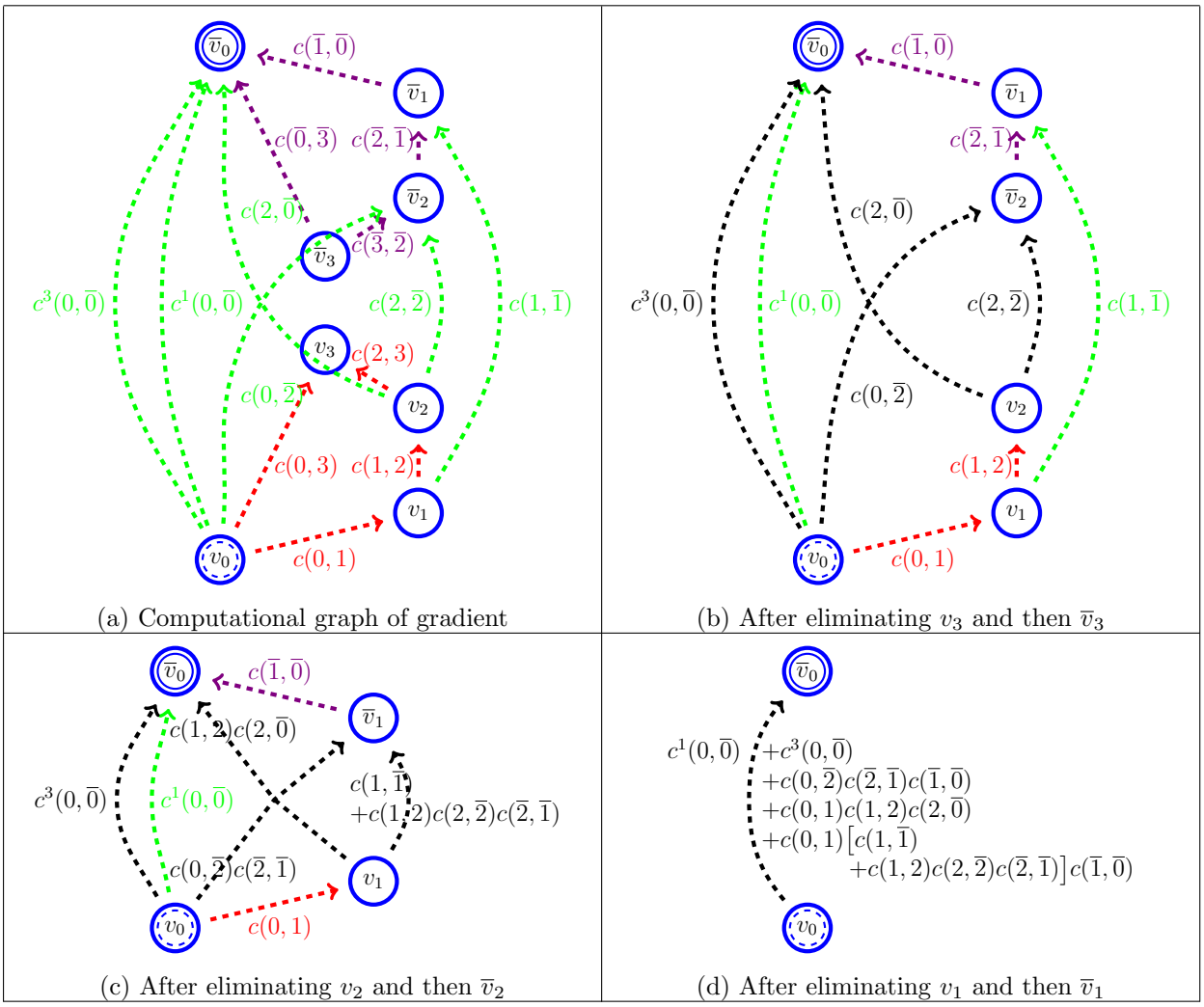


Figure 5: Vertex elimination on the computational graph of gradient with parallel edges.

that the optimal elimination order preserves symmetry [3]. From a practical perspective, preserving symmetry during vertex elimination can reduce the memory requirement and number of updates by storing only half of the graph.

Though it is not an optimal algorithm in theory, the Edge Pushing algorithm is efficient in practice because it can be easily implemented as a reverse sweep of the SAC sequence that sequentially accesses the function trace. The symmetry of the Hessian is fully preserved during the evaluation. Another reason for its efficiency is that an optimized incremental version of the Algorithm 4 can exploit the sparsity in the elemental functions and the Hessian, which leads to reduced time complexity. An expression for this complexity is provided in [2].

Although the Edge Pushing and Vertex Elimination approaches are equivalent in the sense we have proved, the Edge Pushing algorithm is capable of working on objective functions with high arithmetic complexity, because it does not need to explicitly construct the computational graph of the objective function nor the computational graph of the gradient unlike Vertex Elimination. For complex functions, where the function trace could exceed the memory or disk capacity, the Edge Pushing algorithm can be incorporated with check-pointing [9] to make the computation feasible.

6 Conclusion and Future Work

We showed that the Edge Pushing algorithm is equivalent to a variant of the Vertex Elimination algorithm that preserves symmetry and eliminates vertices in a reverse topological order. Although the two algorithms

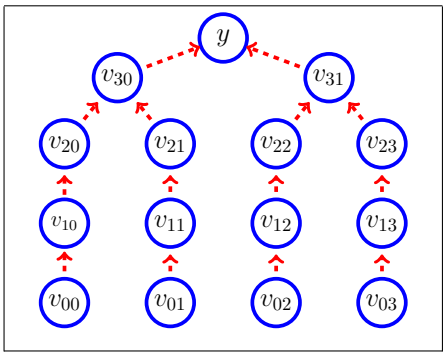


Figure 6: A counter-example where reverse topological order is not optimal when preserving symmetry. We show only the computational graph G of the objective function. The image part representing the gradient and edges joining the two parts are omitted for simplicity.

are derived from different perspectives, their equivalence suggests that there are essential relationships between different AD algorithms evaluating the same derivative. Understanding these relationships could give us insights into how to improve current AD algorithms.

Beside vertex elimination, there are other elimination techniques, e.g, edge elimination and face elimination [5]. They can be viewed as a procedure for accumulating the Jacobian on the computational graph and can be put into a framework in which the Schur complement of a triangular system is evaluated. The triangular system is defined by the computational graph G and the result gives the Jacobian matrix. Following this paradigm, the Hessian evaluation can be considered as computing the Schur complement of an augmented triangular system defined by G_g, the computational graph of the gradient. The work in this paper leads to insights on this extended Schur complement model that could be the scope of further study.

Another interesting future study considers that the Edge Pushing algorithm is based on a second order invariant in reverse mode AD considered in this paper, and this invariant can be extended into any high order derivatives by generalization. Thus we can derive high order reverse mode algorithms by this methodology, and we have done this in related work under submission. It will be interesting to see if there are similar analogies between the high order reverse mode and a form of vertex elimination on a suitably defined graph.

Acknowledgements

This work was supported by the U.S. National Science Foundation grant CCF-1552323, and the Office of Advanced Scientific Computing Research within the Office of Science of the Department of Energy under grant DE-SC0010205.

References

[1] Robert Mansel Gower and Margarida Mello. A New Framework for The Computation of Hessians. *Optimization Methods and Software*, 27(2):251–273, 2012.

[2] Mu Wang, Assefaw Gebremedhin, and Alex Pothen. Capitalizing on Live Variables: New Algorithms for Efficient Hessian Computation via Automatic Differentiation. *Mathematical Programming Computation*, pages 1–41, 2016.

[3] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[4] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, PA, 1991.

[5] Uwe Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of The Chain Rule to Computational Graphs*. PhD thesis, Technische Universitat Dresden, 1999.

[6] Uwe Naumann. Optimal Jacobian Accumulation is NP-complete. *Mathematical Programming*, 112(2):427–441, 2008.

[7] Walter Baur and Volker Strassen. The Complexity of Partial Derivatives. *Theoretical Computer Science*, 22(3):317–330, 1983.

[8] Sanjukta Bhowmick and Paul D Hovland. A Polynomial-Time Algorithm for Detecting Directed Axial Symmetry in Hessian Computational Graphs. In *Advances in Automatic Differentiation*, pages 91–102. Springer, 2008.

[9] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: an Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[10] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, volume 24. SIAM, 2012.

[11] Lawrence CW Dixon. Use of Automatic Differentiation for Calculating Hessians and Newton Steps. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114–125. SIAM, 1991.