Hessian Matrices via Automatic Differentiation

Robert Mansel Gower* Margarida P. Mello†

Institute of Mathematics, Statistics and Scientific Computing
State University of Campinas-UNICAMP

September 29, 2010

ABSTRACT

We investigate the computation of Hessian matrices via Automatic Differentiation, using a graph model and an algebraic model. The graph model reveals the inherent symmetries involved in calculating the Hessian. The algebraic model, based on Griewank and Walther's state transformations [7], synthesizes the calculation of the Hessian as a formula. These dual points of view, graphical and algebraic, lead to a new framework for Hessian computation. This is illustrated by giving a new correctness proof for Griewank and Walther's reverse Hessian algorithm [7, p. 157] and by developing `edge_pushing`, a new truly reverse Hessian computation algorithm that fully exploits the Hessian's symmetry. Computational experiments compare the performance of `edge_pushing` on sixteen functions from the CUTE collection [1] against two algorithms available as drivers of the software ADOL-C [4, 8, 14], and the results are very promising.

1 Introduction

Within the context of nonlinear optimization, algorithms that use variants of Newton's method must repeatedly calculate or obtain approximations of the Hessian matrix or Hessian-vector products. Interior-point methods, ubiquitous in nonlinear solvers [3], fall in this category. For instance, the nonlinear optimization packages LOQO [12] and IPOPT [13] require that the user supply the Hessian, whereas KNITRO [2] is more flexible, but also uses Hessian information of some kind or other. Thus the need to efficiently calculate Hessian matrices is driven by the rising popularity of constraint optimization methods that employ second-order information.

Automatic Differentiation *AD* has had a lot of success in calculating gradients and Hessian-vector products with reverse *AD* procedures[1] that have the same time complexity as that of evaluating the underlying function.

*Partially supported by CNPq and FAPESP (Grant 2009/04785-7). `gowerrobert@gmail.com`

†Partially supported by CNPq-PRONEX Optimization and FAPESP (Grant 2006/53768-0). `margarid@ime.unicamp.br`

[1]Reverse in the sense that the order of evaluation is opposite to the order employed in calculating a function value.

Attempts to efficiently calculate the entire Hessian matrix date back to the work of Jackson and McCormick [10], based on Jackson's dissertation. Their method may be classified as a forward mode routine. It is similar to Griewank's forward mode routine [7], though the latter uses sparsity information in a different way. Despite the fact that [10] mentions plans for future work on the implementation and development of new routines using the ideas introduced, apparently no further reports were published along these lines. Truly effective methods in use combine graph coloring with Hessian-vector AD routines [4, 14].

The paper is organized as follows. Section 2 presents concepts and notation regarding function and gradient evaluation in AD. The graph model for Hessian computation is developed in Section 3 and the algebraic formula for the Hessian is obtained in the next section. This formula is employed in Section 5 to show the correctness of Griewank and Walther's reverse Hessian algorithm [7]. The new algorithm, `edge_pushing`, is described in Section 6. The computational experiments are reported in Section 7 and we close with conclusions and comments on future work.

2 Preliminaries: function and gradient computation

In order to simplify the discussion, we consider functions $f\colon \mathbb{R}^n \to \mathbb{R}$ that are twice continuously differentiable. It is more convenient and the results obtained can be generalized in a straightforward manner to smaller domains and functions that are twice continuously differentiable by parts. There are of course multiple possibilities for expressing a function, but even if one chooses a specific way to write down a function, or a specific way of programming a function f, one still may come up with several distinct translations of f into a finite sequential list of functions. We assume in the following that such a list has already been produced, namely there exists a sequence $(\phi_{1-n}, \ldots, \phi_0, \phi_1, \ldots, \phi_\ell)$, such that the first n functions are the coordinate variables, each *intermediate function* ϕ_i, for $i = 1, \ldots, \ell$, is a function of previous functions in the sequence, and, if we sweep this sequence in a forward fashion, starting with some fixed vector $x = (\phi_{1-n}, \ldots, \phi_0)$, the value obtained for ϕ_ℓ coincides with the value of $f(x)$. Jackson and McCormick [10] dealt with a very similar concept, which they called a *factorable function*, but in that case the intermediate functions were either sums or products of precisely two previous functions, or generic functions of a single previous function, that is, unary functions. Although the framework for calculating the Hessian developed here is valid for intermediate functions with any number of input variables, when specifying the algorithms and evaluating their complexity bounds, we assume that the functions ϕ_i, for $i = 1, \ldots, \ell$, are either unary or binary.

It is very convenient to model the sequential list $(\phi_{1-n}, \ldots, \phi_\ell)$ and the interdependence amongst its components as an acyclic digraph $G = (N, A)$, called *computational graph*. Loosely speaking, the computational graph associated with the list has nodes $\{1-n, \ldots, \ell\}$ and edges $\{(j, i) \mid \phi_i \text{ depends on the value of } \phi_j\}$. The interdependence relations are thus translated into predecessor relations between nodes, and are denoted by the symbol $\prec$. Thus the arc (j, i) embodies the precedence relation $j \prec i$. Notice that, by construction, $j \prec i$ implies $j < i$. Furthermore, if we denote by v_i the output value of ϕ_i for a given input, then we may shorten, for instance, the expression $v_i = \phi_i(v_j, v_k)$ to $v_i = \phi_i(v_j)_{j \prec i}$. Figure 1 shows the computational graph of function $f(x) = (x_{-1}x_0)(x_{-1} + x_0)$ that corresponds to the sequence $(\phi_{-1}, \ldots, \phi_3) = (x_{-1}, x_0, x_{-1}x_0, x_{-1} + x_0, (x_{-1}x_0)(x_{-1} + x_0))$.

Due to the choice of the numbering scheme for the ϕ's, commonly adopted in the literature, we

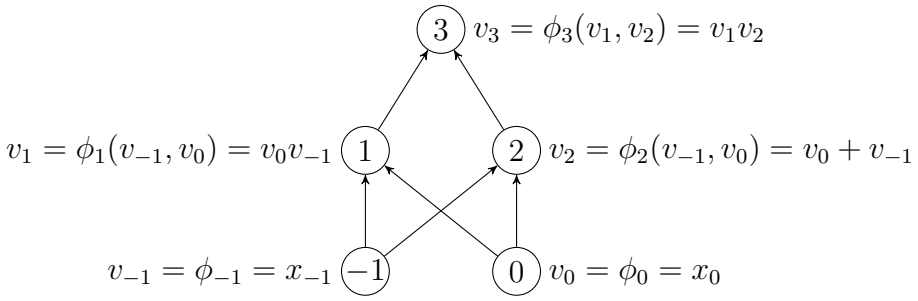


Figure 1: Computational graph of the function $f(x) = (x_{-1}x_0)(x_{-1} + x_0)$.

found it convenient to apply, throughout this article, a shift of $-n$ to the indices of all matrices and vectors. We already have $x \in \mathbb{R}^n$, which, according to this convention, has components x_{1-n}, x_{2-n}, ..., x_0. Similarly, the rows/columns of the Hessian f'' are numbered $1-n$ through 0. Other vectors and matrices will be gradually introduced, as the need arises for expressing and deducing mathematical properties enjoyed by the data.

For instance, a forward sweep through the sequential list $(\phi_{1-n}, \ldots, \phi_\ell)$ could be recorded in a sequence of $(n+\ell)$-dimensional vectors that accumulate the calculations up to a point, say

$$\mathrm{v}^i = (v_{1-n}, \ldots, v_i, 0, \ldots, 0)^T, \quad \text{for } i = 0, \ldots, \ell. \tag{1}$$

Then the computation of v_i corresponds to applying the *state transformation*

$$\begin{aligned} \Phi_i \quad : \quad & \mathbb{R}^{n+\ell} \to \mathbb{R}^{n+\ell} \\ & y \quad \mapsto (y_{1-n}, \ldots, y_{i-1}, \phi_i(y_j)_{j \prec i}, y_{i+1}, \ldots, y_\ell)^T \end{aligned} \tag{2}$$

to vector v^{i-1} and we arrive at Griewank and Walther's [7] representation of f as a composition of state transformations

$$f(x) = e_\ell^T \Phi_\ell \circ \Phi_{\ell-1} \circ \cdots \circ \Phi_1(P^T x), \tag{3}$$

where the $n \times (n+\ell)$ matrix P is zero except for the leftmost n-dimensional block, which contains an identity matrix.

The advantage of vector/matrix notation is that formulas expressed in terms of vector/matrix operations usually lend themselves to straightforward algorithmic implementations. Nevertheless, when analyzing complexity issues and actual implementation, one has to translate block operations with vectors or matrices into componentwise operations on individual variables.

This is illustrated in Algorithms 1 and 2, which express the sequence of operations executed in a forward sweep of the sequential list of ϕ's, first as operations on single variables and then as operations on vectors. Regarding the output of Algorithm 2, notice that, according to our convention, the canonical vector e_ℓ has the first $n+\ell-1$ components equal to zero and the last equal to 1.

Algorithm 1: Componentwise evaluation of function f with computational graph G.

Input: $x \in \mathbb{R}^n$, computational graph G
for $i = 1-n, \ldots, 0$ **do**
  $v_i = x_i$
end
for $i = 1, \ldots, \ell$ **do**
  $v_i = \phi_i(v_j)_{j \prec i}$
end
Output: v_ℓ

Algorithm 2: Block evaluation of function f with computational graph G.

Input: $x \in \mathbb{R}^n$, computational graph G
for $i = 1-n, \ldots, 0$ **do**
  $v_i = x_i$
end
for $i = 1, \ldots, \ell$ **do**
  $\mathrm{v}^i = \Phi_i(\mathrm{v}^{i-1})$
end
Output: $e_\ell^T \mathrm{v}_\ell$

A very convenient consequence of formula (3) is that it, in turn, provides a closed formula for the gradient of f, using the chain rule recursively,

$$(\nabla f(x))^T = e_\ell^T \Phi_\ell' \Phi_{\ell-1}' \cdots \Phi_1'(P^T x) P^T, \tag{4}$$

which can be used to deduce (or justify) algorithms for calculating $\nabla f(x)$.

One possibility is to calculate the product in (4) in a left-to-right fashion. In this case one must start at the last Jacobian Φ_ℓ'. Its computation involves obtaining the derivatives of ϕ_ℓ from a table and substituting the appropriate values for its arguments. The latter are the values associated with the predecessors of node ℓ. Analogously, the values associated with predecessors of nodes $\ell-1$, $\ell-2$, etc., must be known in order to compute the Jacobians $\Phi_{\ell-1}'$, $\Phi_{\ell-2}'$, etc. This means that a forward sweep of the computational graph must already have been performed, and the v_i's (or, v_i's) stored for later use. We shall call the data structure that contains all information concerning the function evaluation produced during the foward sweep a *tape* $\mathcal{T}$, borrowing the naming convention of [8]. Thus the tape contains the relevant recordings of a forward sweep along with the computational graph of f.

Algorithm 3: Block reverse evaluation of ∇f.

Input: tape $\mathcal{T}$
initialization: $\overline{\mathrm{v}} = e_\ell$
for $i = \ell, \ldots, 1$ **do**
  $\overline{\mathrm{v}}^T = \overline{\mathrm{v}}^T \Phi_i'$
end
Output: $\nabla f = P\overline{\mathrm{v}}$

Algorithm 3 implements the left-to-right product computation. The necessary partial products are stored in $\overline{\mathrm{v}}$, and, right before node i is swept, the vector $\overline{\mathrm{v}}$ satisfies

$$\overline{\mathrm{v}}^T = e_\ell^T \prod_{j=1}^{\ell-i} \Phi_{\ell-j+1}'. \tag{5}$$

The translation to a componentwise computation of the vector-matrix products of Algorithm 3 is very much simplified by the special structure of the Jacobian of the state transformation Φ_i. This

follows from the fact that the function in component j of Φ_i is given by

$$[\Phi_i]_j(y) = \begin{cases} y_j, & \text{if } j \neq i, \\ \phi_i(y_j)_{j \prec i}, & \text{if } j = i. \end{cases} \tag{6}$$

Since row j of the Jacobian Φ_i' is the transposed gradient of $[\Phi_i]_j$, we arrive at the following block structure for Φ_i':

$$\Phi_i' = \left[\begin{array}{c|c|c} & & \\ I & 0 & 0 \\ & & \\ \hline & & \\ c^T & 0 & 0 \\ & & \\ \hline & & \\ 0 & 0 & I \\ & & \end{array}\right] \begin{array}{l} 1-n \\ \vdots \\ i-1 \\ \text{row } i, \\ i+1 \\ \vdots \\ \ell \end{array} \tag{7}$$

where

$$c_j = \frac{\partial \phi_i}{\partial v_j}, \quad \text{for } j = 1-n, \ldots, i-1. \tag{8}$$

Thus c^T is basically the transposed gradient of ϕ_i padded with the convenient number of zeros at the appropriate places. In particular, it has at most as many nonzeros as the number of predecessors of node i, and the post-multiplication $\overline{\mathrm{v}}^T \Phi_i'$ affects the components of $\overline{\mathrm{v}}$ associated with the predecessors of node i and zeroes component i. In other words, denoting component i of $\overline{\mathrm{v}}$ by $\bar{v}_i$, the block assignment in Algorithm 4 is equivalent to

$$\bar{v}_j \leftarrow \begin{cases} \bar{v}_j + \bar{v}_i \dfrac{\partial \phi_i}{\partial v_j}, & \text{if } j \prec i, \\ 0, & \text{if } j = i, \\ \bar{v}_j, & \text{otherwise.} \end{cases}$$

Now this assignment is done as the node i is swept, and, therefore, in subsequent iterations component i of $\overline{\mathrm{v}}$ will not be accessed, since the loop visits nodes in decreasing index order. Hence setting component i to zero has no effect on the following iterations. Eliminating this superfluous reduction, we arrive at Algorithm 3, the componentwise (slightly altered) version of Algorithm 3.

Algorithm 4: Componentwise reverse evaluation of ∇f.

Input: tape $\mathcal{T}$
initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{\ell-1} = 0$, $\bar{v}_\ell = 1$
for $i = \ell, \ldots, 1$ **do**
  for $j \prec i$ **do**
    $\bar{v}_j += \bar{v}_i \dfrac{\partial \phi_i}{\partial v_j}$
  end
end
Output: $\nabla f = (\bar{v}_{1-n}, \ldots, \bar{v}_0)^T$

In order to give a graph interpretation of Algorithm 4, let $c_{ji} = \partial \phi_i / \partial v_j$ be the weight of arc (j, i), and define the weight of a directed path from node j to node k as the product of the weights of

the arcs in the path. Then, one can easily check by induction that, right before node i is swept, the *adjoint* $\bar{v}_i$ contains the sum of the weights of all the paths from node i to node ℓ. As node i is swept, the value of $\bar{v}_i$ is properly distributed amongst its predecessors, in the sense that, accumulated in $\bar{v}_j$, for each predecessor j, is the contribution of all paths from j to ℓ that contain node i, weighted by c_{ji}. Hence, at the end of Algorithm 4, the adjoint $\bar{v}_{i-n}$, for $i = 1, \ldots, n$, contains the sum of the weights of all paths from $i - n$ to ℓ. This is in perfect accordance with the explanation for the computation of partial derivatives given in some Calculus textbooks, see, for instance, [11, p. 940].

Figure 2 illustrates the modification of the $\bar{v}$'s after the sweeping of node 2 of the computational graph in Figure 1. Indicated beside node i is the value v_i obtained in the forward sweep that preceded the gradient computation and the current value of $\bar{v}_i$. We assume the values of the input variables were $x_{-1} = a$ and $x_0 = b$. As node i is swept in the backwards sweep, the weights of the arcs incident thereto are calculated. Appended to the arcs in the figure are the weights that have already been calculated.

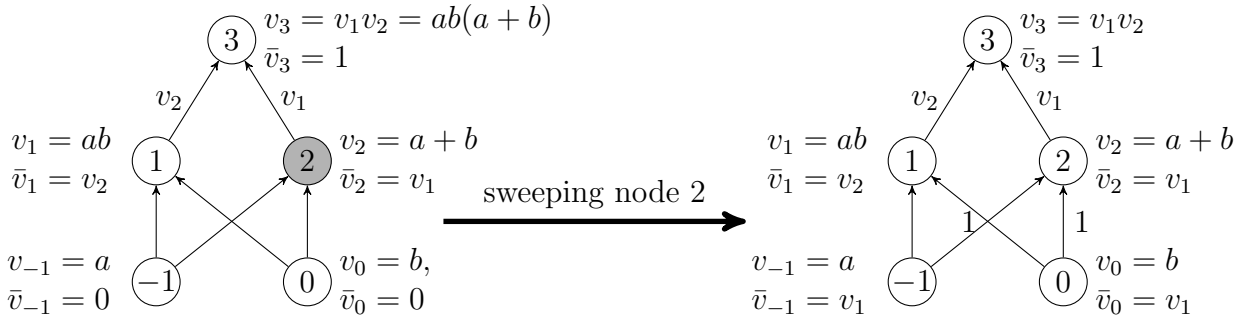


Figure 2: $\bar{v}$'s values updated as node 2 is swept.

Of course different ways of calculating the product of the Jacobians of the state transformations in (4) may give rise to different algorithms. Thus the state transformation point of view, allied with the chain rule, produced a closed formula for the gradient that opened the door to the development of a host of algorithms for calculating the gradient. In the following, using the same ingredients, we obtain a closed formula for the Hessian, and show how it can be used to justify known algorithms as well as suggest a new algorithm for Hessian computation. Before that, however, we develop a graph understanding of the Hessian computation.

3 Hessian via computational graph

Creating a graph model for the Hessian is also very useful, as it provides insight and intuition regarding the workings of Hessian algorithms. Not only can the graph model suggest algorithms, it can also be very enlightening to interpret the operations performed by an algorithm as operations on variables associated with the nodes and arcs of a computational graph.

Since second order derivatives are simply first order derivatives of the gradient, a natural approach to their calculation would be to build a computational graph for the gradient and apply Algorithm 4 to this new graph. We do this to build our understanding of the problem, but later on we will see that it is possible to work with the original graph plus some new edges.

Of course the gradient may be represented by distinct computational graphs, or equivalently, sequential lists of functions, but the natural one to consider is the one associated with the com-

putation performed by Algorithm 4. Assuming this choice, the gradient $\nabla f = (\bar{v}_{1-n}, \ldots, \bar{v}_0)^T$ is a composite function of $(\bar{v}_1, \ldots, \bar{v}_\ell)$, as well as $(v_{1-n}, \ldots, v_\ell)$, which implies that the gradient (computational) graph $G^g = (N^g, A^g)$ must contain G. The graph G^g is basically built upon G by adding nodes associated with $\bar{v}_i$, for $i = 1-n, \ldots, \ell$, and edges representing the functional dependencies between these nodes.

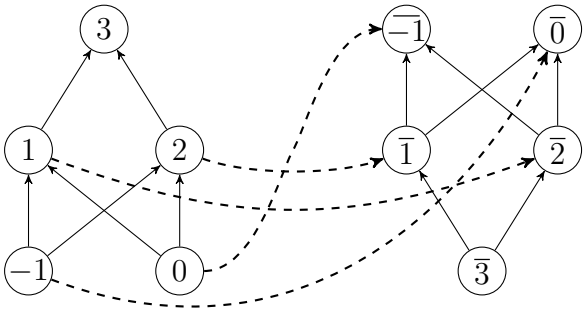
Thus the node set N^g contains $2(n+\ell)$ nodes $\{1-n, \ldots, \ell, \overline{1-n}, \ldots, \bar{\ell}\}$, the first half associated with the original variables and the second half with the adjoint variables. The arc set is $A^g = A_1 \cup A_2 \cup A_3$, where A_1 contains arcs with both endpoints in "original" nodes; A_2, arcs with both endpoints in "adjoint" nodes and A_3, arcs with endpoints of mixed nature. Since running Algorithm 4 does not introduce new dependencies amongst the original v's, we have that $A_1 = A$.

The new dependent variables created by running Algorithm 4 satisfy

$$\bar{v}_i = \sum_{k|i\prec k} \bar{v}_k \frac{\partial v_k}{\partial v_i} \tag{9}$$

at the end of the algorithm. This means the set of successors of node i gives rise to a corresponding set of predecessors of adjoint node $\bar{\imath}$. Therefore arcs in A_2 are copies of the arcs in A with the orientation reversed. The graph G^g thus contains G and a kind of a mirror copy of G.

Arcs in A_3 arise from the partial derivatives. Since none of the original variables depends on the adjoint ones, arcs in A_3 are of the type $(k, \bar{\imath})$. They arise from the partial derivatives appearing in the sum (9). There is an arc $(k, \bar{\imath})$ only if $\partial v_j/\partial v_i$ is a function of v_k, for $k \succ i$. Since this can only be the case if $k \prec j$, we have that $(k, \bar{\imath}) \in A_3$ only if k and i have a common successor in G. Thus the function $\bar{v}_i = \phi_{\bar{\imath}}$ associated with node $\bar{\imath}$, given in (9), is a function of $\bar{v}_k$, for k's that are sucessors of i and, potentially, of v_k for k's that share a common successor with i. The only exception is the function associated with node $\bar{\ell}$, since $\bar{v}_\ell \equiv 1$. Figure 3 shows the computational graph of the gradient of the function f given in Figure 1. Notice that on the left we have the computational graph of f and, on the right, a mirror copy thereof. Arcs in A_3 are the ones drawn dashed in the picture. This graph has already been obtained in [7, p. 237].



$$\begin{array}{l|l} v_{-1} = x_{-1} & \bar{v}_3 = \phi_{\bar{3}} = 1 \\ v_0 = x_0 & \bar{v}_2 = \phi_{\bar{2}} = \bar{v}_3 v_1 \\ v_1 = \phi_1 = v_{-1} v_0 & \bar{v}_1 = \phi_{\bar{1}} = \bar{v}_3 v_2 \\ v_2 = \phi_2 = v_{-1} + v_0 & \bar{v}_0 = \phi_{\bar{0}} = \bar{v}_2 + \bar{v}_1 v_{-1} \\ v_3 = \phi_3 = v_1 + v_2 & \bar{v}_{-1} = \phi_{\overline{-1}} = \bar{v}_2 + \bar{v}_1 v_0 \end{array}$$

Figure 3: Gradient computational graph G^g of the function $f(x) = (x_{-1}x_0)(x_{-1}+x_0)$, represented by the computational graph in Figure 1.

We conclude that

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{p|\text{path from } i \text{ to } \bar{\jmath}} \text{weight of path } p, \tag{10}$$

where the weight of path p is simply the product of the weights of the arcs in p.

The weights of arcs $(i,j) \in A_1$ are already know. Equation (9) implies that the weight of arc $(\bar{\imath},\bar{\jmath}) \in E_2$ is

$$c_{\bar{\imath}\bar{\jmath}} = \frac{\partial v_i}{\partial v_j} = c_{ji}, \tag{11}$$

that is, arc (i,j) has the same weight as its mirror image.

The weight of arc $(k,\bar{\imath})$ is also obtained from (9)

$$\begin{aligned} c_{j\bar{\imath}} &= \sum_{k|i\prec k} \bar{v}_k \frac{\partial^2 v_k}{\partial v_j \partial v_i} \\ &= \sum_{k|i\prec k \text{ and } j\prec k} \bar{v}_k \frac{\partial^2 v_k}{\partial v_j \partial v_i}, \end{aligned} \tag{12}$$

since the partial derivative $\partial^2 v_k/\partial v_j \partial v_i$ is identically zero if k is not a successor of j. In particular, (12) and the fact that f is twice continuously differentiable implies that

$$c_{j\bar{\imath}} = c_{i\bar{\jmath}}, \text{ for } j \neq i. \tag{13}$$

Notice that arcs in A_3 are the only ones with second-order derivatives as weights. In a sense, they carry the nonlinearity of f, which suggests the denomination *nonlinear arcs*.

Regarding the paths in G^g from i to $\bar{\jmath}$, for fixed $i,j \in \{1-n,\ldots,0\}$, each of them contains a unique nonlinear arc, since none of the original nodes is a successor of an adjoint node. Therefore, the sum in (10) may be partitioned according to the nonlinear arc utilized by the paths as follows:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{(r,\bar{s})\in A_3} \left[\left(\sum_{p|\text{path from } i \text{ to } r} \text{weight of path } p \right) c_{r\bar{s}} \left(\sum_{q|\text{path from } \bar{s} \text{ to } \bar{\jmath}} \text{weight of path } q \right) \right], \tag{14}$$

which reduces to

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{(r,\bar{s})\in A_3} \left[\left(\sum_{p|\text{path from } i \text{ to } r} \text{weight of path } p \right) c_{r\bar{s}} \left(\sum_{q|\text{path from } j \text{ to } s} \text{weight of path } q \right) \right], \tag{15}$$

using the symmetry in (11).

On close examination, there is a lot of redundant information in G^g. One really doesn't need the mirror copy of G, since the information attached to the adjoint nodes can be recorded associated to the original nodes and the arc weights of the mirror arcs are the same. Now if we fold back the mirror copy over the original, identifying nodes k and $\overline{k}$, we obtain a graph with same node set as G but with an enlarged set of arcs. Arcs in A will be replaced by pairs of arcs in opposite directions and nonlinear arcs will become either loops (in case one had an arc $(i,\bar{\imath})$ in A_3) or pairs of arcs with opposite orientations between the same pair of nodes. Also, equations (11) and (13) imply that all arcs in parallel have the same weight, see Figure 4. This is still too much redundancy. We may leave arcs in A as they are, replace the pairs of directed nonlinear arcs in opposite directions with a single nondirected arc between the same pair of nodes and the directed loops by undirected ones,

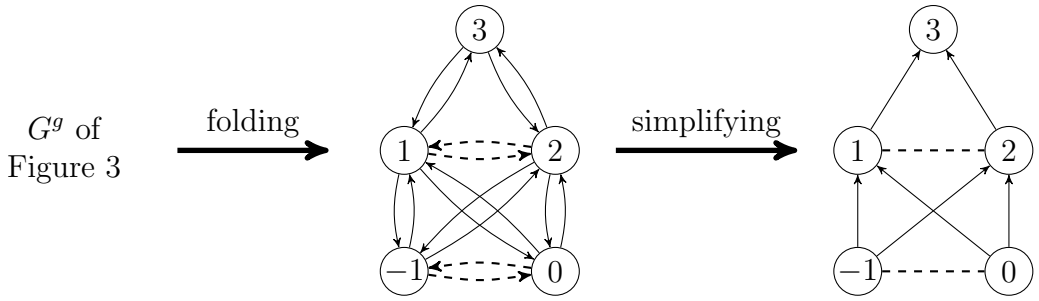


Figure 4: Folding of the gradient computational graph of Figure 3 and further elimination of redundancies.

as exemplified in Figure 4, as long as we keep in mind the special characteristics of the paths we're interested in.

The paths needed for the computation of the Hessian, in the folded and simplified graph, are divided into three parts. In the first part we have a directed path from some zero in-degree node, say i, to some other node, say r. Next comes an undirected nonlinear arc (r, s). The last part is a path from s to another zero in-degree node, say j, in which all arcs are traveled in the wrong direction. Of course, both the first and third parts of the path may be empty, only the middle part (the nonlinear arc) is mandatory.

This folded and simplified graph can be interpreted as a reduced gradient graph, with the symmetric redundancies removed. The graph together with the tri-parted path interpretation for partial derivatives constitutes our graph model for the Hessian. In Section 6 we will present an algorithm that takes full advantage of theses symmetries and has a natural interpretation as an algorithm that accumulates the weights of these special paths on this graph.

4 Hessian formula

The closed formula to be developed concerns the Hessian of a function g that is defined as a linear combination of the functions $\Psi_1, \ldots, \Psi_p$, or, in matrix form,

$$g(x) = y^T \Psi(x), \tag{16}$$

where $y \in \mathbb{R}^p$ and $\Psi \in C^2(\mathbb{R}^n, \mathbb{R}^p)$. The linearity of the differential operator implies that the Hessian of g is simply the linear combination of the Hessians of $\Psi_1, \ldots, \Psi_p$:

$$g''(x) = \sum_{i=1}^{p} y_i \Psi_i''(x). \tag{17}$$

This motivates the introduction of the following definition of the vector-tensor product $y^T \Psi''(x)$, in order to establish an analogy between the linear combinations in (16) and (17):

$$g''(x) = (y^T \Psi(x))'' = y^T \Psi'' = \sum_{i=1}^{p} y_i \Psi_i''(x). \tag{18}$$

Next we need to establish how to express g'' when Ψ is a composition of vector functions of several variables, the subject of the next Proposition.

Proposition 1 *Let $y \in \mathbb{R}^p$, $\Omega \in C^2(\mathbb{R}^n, \mathbb{R}^m)$, $\Theta \in C^2(\mathbb{R}^m, \mathbb{R}^p)$ and $\Psi(x) = \Theta \circ \Omega(x)$. Then*

$$y^T \Psi'' = (\Omega')^T (y^T \Theta'') \Omega' + (y^T \Theta') \Omega''. \tag{19}$$

Proof. By definition, applying differentiation rules, and using the symmetry of the Hessian, we may calculate entry (j,k) of the Hessian as follows:

$$\begin{aligned}
(y^T \Psi''(x))_{jk} &= \sum_i y_i \frac{\partial^2 \Psi_i(x)}{\partial x_j \partial x_k} \\
&= \sum_i y_i \frac{\partial}{\partial x_j} \left(\frac{\partial \Theta_i(\Omega(x))}{\partial x_k} \right) \\
&= \sum_i y_i \frac{\partial}{\partial x_j} \left(\sum_{r=1}^{m} \frac{\partial \Theta_i(\Omega(x))}{\partial \Omega_r} \frac{\partial \Omega_r(x)}{\partial x_k} \right) \\
&= \sum_i \sum_r y_i \left[\frac{\partial}{\partial x_j} \left(\frac{\partial \Theta_i(\Omega(x))}{\partial \Omega_r} \right) \right] \frac{\partial \Omega_r(x)}{\partial x_k} + \sum_i \sum_r y_i \frac{\partial \Theta_i(\Omega(x))}{\partial \Omega_r} \frac{\partial^2 \Omega_r(x)}{\partial x_j \partial x_k} \\
&= \sum_r \sum_s \sum_i y_i \frac{\partial^2 \Theta_i(\Omega(x))}{\partial \Omega_s \partial \Omega_r} \frac{\partial \Omega_s(x)}{\partial x_j} \frac{\partial \Omega_r(x)}{\partial x_k} + \sum_r (y^T \Theta'(\Omega(x))_r (\Omega_r''(x))_{jk} \\
&= \sum_s \sum_r (y^T \Theta''(\Omega(x)))_{rs} \left(\Omega'(x)\right)_{sj} \left(\Omega'(x)\right)_{rk} + \sum_r (y^T \Theta'(\Omega(x))_r (\Omega_r''(x))_{jk} \\
&= \sum_s \left(\Omega'(x)\right)_{sj} \sum_r (y^T \Theta''(\Omega(x)))_{sr} \left(\Omega'(x)\right)_{rk} + \sum_r (y^T \Theta'(\Omega(x))_r (\Omega_r''(x))_{jk}, \\
&= \left((\Omega(x))^T (y^T \Theta''(\Omega(x))) \Omega'(x) \right)_{jk} + \left((y^T \Theta'(\Omega(x))) \Omega''(x) \right)_{jk},
\end{aligned}$$

which is the entry (j,k) of the right-hand-side of (19). ∎

Although we want to express the Hessian of a composition of state transformations, it is actually easier to obtain the closed form for the composition of generic vector multivariable functions, our next result.

Proposition 2 *Let $\Psi_i(x) \in C^2(\mathbb{R}^{m_{i-1}}, \mathbb{R}^m)$, for $i = 1, \ldots, k$, $y \in \mathbb{R}^{m_k}$ and*

$$g(x) = y^T \Psi_k \circ \cdots \circ \Psi_1(x).$$

Then

$$g'' = \sum_{i=1}^{k} \left(\prod_{j=1}^{i-1} (\Psi_j')^T \right) ((\overline{\mathrm{w}}^i)^T \Psi_i'') \left(\prod_{j=1}^{i-1} \Psi_{i-j}' \right), \tag{20}$$

where

$$(\overline{\mathrm{w}}^i)^T = y^T \prod_{j=1}^{k-i} \Psi_{k-j+1}', \quad \text{for } i = 1, \ldots, k. \tag{21}$$

Proof. The proof is by induction on k. When $k = 1$, the result is trivially true, since in this case (20)–(21) reduce to $(\overline{\mathrm{w}}^1)^T\Psi_1'' = y^T\Psi_1''$, which denotes, according to (17), the Hessian of g.

Assume the proposition is true when g is the composition of $k-1$ functions. Now simply rewrite the composition of k functions as follows

$$g = y^T\Psi_k \circ \cdots \circ \Psi_3 \circ \Psi, \tag{22}$$

where $\Psi = \Psi_2 \circ \Psi_1$. Then, applying the induction hypothesis to (22), we obtain

$$g'' = (\Psi')^T \left[\sum_{i=3}^{k} \left(\prod_{j=3}^{i-1}(\Psi_j')^T\right) ((\overline{\mathrm{w}}^i)^T\Psi_i'') \left(\prod_{j=3}^{i-1}\Psi_{i-j}'\right)\right] \Psi' + (\overline{\mathrm{w}}^2)^T\Psi''. \tag{23}$$

The last term in (23) is calculated separetely, using the induction hypothesis, (19) and (21):

$$\begin{aligned} (\overline{\mathrm{w}}^2)^T\Psi'' &= (\Psi_1')^T((\overline{\mathrm{w}}^2)^T\Psi_2'')\Psi_1' + ((\overline{\mathrm{w}}^2)^T\Psi_2')\Psi_1'' \\ &= (\Psi_1')^T((\overline{\mathrm{w}}^2)^T\Psi_2'')\Psi_1' + (\overline{\mathrm{w}}^1)^T\Psi_1''. \end{aligned} \tag{24}$$

Using the fact that $\Psi' = \Psi_2'\Psi_1'$, and expression (24) obtained for the last term, (23) becomes

$$\begin{aligned} g'' &= (\Psi_1')^T(\Psi_2')^T \left[\sum_{i=3}^{k} \left(\prod_{j=3}^{i-1}(\Psi_j')^T\right) ((\overline{\mathrm{w}}^i)^T\Psi_i'') \left(\prod_{j=3}^{i-1}\Psi_{i-j}'\right)\right] \Psi_2'\Psi_1' \\ &\quad +(\Psi_1')^T((\overline{\mathrm{w}}^2)^T\Psi_2'')\Psi_1' + (\overline{\mathrm{w}}^1)^T\Psi_1'' \\ &= \sum_{i=1}^{k} \left(\prod_{j=1}^{i-1}(\Psi_j')^T\right) ((\overline{\mathrm{w}}^i)^T\Psi_i'') \left(\prod_{j=1}^{i-1}\Psi_{i-j}'\right), \end{aligned}$$

which completes the proof. ∎

The Hessian of the composition of state transformations follows easily from Proposition 2.

Corollary 3 *Let f be the composition of state transformations given in (3). Then its Hessian is*

$$f'' = P\sum_{i=1}^{\ell} \left(\prod_{j=1}^{i-1}(\Phi_j')^T\right) ((\overline{\mathrm{v}}^i)^T\Phi_i'') \left(\prod_{j=1}^{i-1}\Phi_{i-j}'\right) P^T, \tag{25}$$

where

$$(\overline{\mathrm{v}}^i)^T = e_\ell^T \prod_{j=1}^{\ell-i}\Phi_{\ell-j+1}', \quad \textit{for } i = 1, \ldots, \ell. \tag{26}$$

Proof. Simply apply (20) to the composition of $\ell + 1$ functions, where $\Psi_i = \Phi_i$, for $i = 1, \ldots, \ell$, $\Psi_0(x) = P^Tx$, and use the facts that $\Psi_0' = P^T$ and $\Psi_0'' = 0$. ∎

The expression for the Hessian of f can be further simplified by noting that the tensor Φ_i'' is null except for the Hessian of its component i, $[\Phi_i]_i$, since the other components are just projections onto a single variable, see (6). Thus the vector-tensor product in (25) reduces to

$$(\overline{\mathrm{v}}^i)^T\Phi_i'' = \bar{v}_i^i[\Phi_i]_i'', \tag{27}$$

where $\overline{\mathrm{v}}^i = (\bar{v}^i_{1-n}, \ldots, \bar{v}^i_\ell)$. Furthermore, notice that $[\Phi_i]''_i$ is just the Hessian of ϕ_i padded with the appropriate number of zeros at the right places.

Letting

$$\dot{V}^i = \left(\prod_{j=1}^{i-1} \Phi'_{i-j} \right) P^T, \quad \text{for } i = 1, \ldots, \ell, \tag{28}$$

and using (27), (25) reduces to

$$\begin{aligned} f'' &= \sum_{i=1}^{\ell} (\dot{V}^i)^T \bar{v}^i_i [\Phi_i]''_i \dot{V}^i \\ &= \sum_{i=1}^{\ell} \bar{v}^i_i (\dot{V}^i)^T [\Phi_i]''_i \dot{V}^i. \end{aligned} \tag{29}$$

5 Griewank and Walther's reverse Hessian algorithm

Although the inception of Griewank and Walther's reverse Hessian computation algorithm [7, p.157], presented in block form in Algorithm 5, follows a different line of reasoning, its correctness may be established by means of (25).

Algorithm 5: Griewank and Walther's Reverse Hessian computation algorithm.

Input: a tape $\mathcal{T}$
initialization: $\overline{\mathrm{v}} = e_\ell$, $W = 0$, $\dot{V}^1 = P^T$
for $i = 2, \ldots, \ell$ **do**
  $\dot{V}^i = \Phi'_{i-1} \dot{V}^{i-1}$
end
for $i = \ell, \ldots, 1$ **do**
  $W = (\Phi'_i)^T W$
  $W += \overline{\mathrm{v}}^T \Phi''_i \dot{V}^i$
  $\overline{\mathrm{v}}^T = \overline{\mathrm{v}}^T \Phi'_i$
end
Output: $f'' = PW$

Algorithm 5 is the translation to block operations, using our notation, of Griewank and Walther's reverse Hessian computation algorithm. It recursively builds parts of expression (25) and then combines them appropriately. It is straightforward to see that $(\dot{V}^1, \ldots, \dot{V}^\ell)$ constructed in the first (forward) loop satisfies (28).

In the second loop the indices are visited in reverse order, or equivalently, a backward sweep of the computational graph is performed. Notice that the computation of $\overline{\mathrm{v}}$ is the same as in Algorithm 3. Thus, recalling (5), at the beginning of the iteration where node i is swept, this vector contains the partial product $e_\ell^T \Phi'_\ell \cdots \Phi'_{i-1}$. Hence, at the iteration where node i is swept, W is incremented by

$$\overline{\mathrm{v}} \Phi''_i \dot{V}^i = ((\overline{\mathrm{v}}^i)^T \Phi''_i) \left(\prod_{j=1}^{i-1} \Phi'_{i-j} \right) P^T.$$

Finally, taking into account the pre-multiplication done at the beginning of the reverse loop, it can be shown by induction that, at the end of the iteration where node i is swept, we have

$$W = \sum_{k=i}^{\ell} \left(\prod_{j=i}^{k-1} (\Phi'_j)^T \right) ((\overline{\mathrm{v}}^k)^T \Phi''_k) \left(\prod_{j=1}^{k-1} \Phi'_{k-j} \right) P^T.$$

This implies that, at the end of the algorithm, PW is precisely the expression for the Hessian of f in (25).

As far as we can ascertain, there are no reports on the implementation and testing of this algorithm. Although the special structure of the Jacobians and Hessians of the state transformations lead to simple and efficient componentwise versions of the block assignments, there are two obvious downsides to this approach of calculating the Hessian. First is the fact that its symmetry is not exploited, and second, $(\dot{V}^1, \ldots, \dot{V}^\ell)$, calculated in the forward loop, needs to be recorded for later use in the second loop, which is potentially a large quantity of memory, even if one takes advantage of its special structure. Finally, the adjective "reverse" is not fully appropriate, since part of the computation must be carried out in the forward loop.

6 A new Hessian computation algorithm: `edge_pushing`

6.1 Development

Expression (25) leads in a straightforward fashion to an algorithm, especially if we think in terms of block operations. Simply put, the sum in (25) is accumulated in a backward sweep as follows. The input is the tape $\mathcal{T}$, so at the iteration where node i is swept, values of ϕ_j, for all j, are known. The updating of the initially null square matrix W of order $n + \ell$ is as follows

$$\begin{array}{rl}
\text{Node } \ell & W \leftarrow (\Phi'_\ell)^T W \Phi'_\ell \\
& W \leftarrow W + (\overline{\mathrm{v}}^\ell)^T \Phi''_\ell \\
\text{Node } \ell - 1 & W \leftarrow (\Phi'_{\ell-1})^T W \Phi'_{\ell-1} \\
& W \leftarrow W + (\overline{\mathrm{v}}^{\ell-1})^T \Phi''_{\ell-1} \\
& \vdots \\
\text{Node } i & W \leftarrow (\Phi'_i)^T W \Phi'_i \\
& W \leftarrow W + (\overline{\mathrm{v}}^i)^T \Phi''_i.
\end{array}$$

Thus, the value of W at the end of the iteration where node i is swept is given by

$$W = \sum_{k=i}^{\ell} \left(\prod_{j=i}^{k-1} (\Phi'_j)^T \right) ((\overline{\mathrm{v}}^k)^T \Phi''_k) \left(\prod_{j=1}^{k-i} \Phi'_{k-j} \right).$$

Notice that, at the iteration where node i is swept, both assignments involve derivatives of Φ_i, which are available. The other piece of information needed is the vector $\overline{\mathrm{v}}^i$, which we know how to calculate via a backward sweep from Algorithm 3. Putting these two together, we arrive at Algorithm 6.

The first striking characteristic of Algorithm 6 is that the symmetry of matrix W is preserved throughout, which is something we can take advantage of in the translation to the componentwise version. Another welcome feature is that only contributions that really matter to f'' are kept and subsequently used, in contrast with other algorihtms, e.g. Algorithm 5, in which one accumulates (possibly linear) dependencies in the $n \times (n+\ell)$ matrix $\dot{V}$, amounting to (possibly a lot) of unused information.

Algorithm 6: Block form of `edge_pushing`.

Input: a tape $\mathcal{T}$
initialization: $\overline{\mathrm{v}} = e_\ell$, $W = 0$
for $i = \ell, \ldots, 1$ **do**
  $W = (\Phi_i')^T W \Phi_i'$
  $W += \overline{\mathrm{v}}^T \Phi_i''$
  $\overline{\mathrm{v}}^T = \overline{\mathrm{v}}^T \Phi_i'$
end
Output: $f'' = PWP^T$

Before delving into the componentwise version of Algorithm 6, there is a key observation to be made about matrix W, established in the following proposition.

Proposition 4 *At the end of the iteration at which node i is swept in Algorithm 6, for all i, the nonnull elements of W lie in the upper diagonal block of size $n + i - 1$.*

Proof. Consider the first iteration, at which node ℓ is swept. At the beginning W is null, so the first block assignment $((\Phi_\ell')^T W \Phi_\ell')$ does not change that. Now consider the assignment

$$W \leftarrow W + (\overline{\mathrm{v}}^\ell)^T \Phi_\ell''.$$

Using (27) and the initialization of $\overline{\mathrm{v}}$, we have

$$(\overline{\mathrm{v}})^T \Phi_\ell'' = \bar{v}_\ell [\Phi_\ell]_\ell'' = [\Phi_\ell]_\ell'',$$

and, since $[\Phi_\ell]_\ell(y) = \phi_\ell(y_j)_{j \prec \ell}$, the nonnull entries of $[\Phi_\ell]_\ell''$ must have column and row indices that correspond to predecessors of node ℓ. This means the last row and column, of index ℓ, are zero. Thus the statement of the proposition holds after the first iteration.

Suppose by induction that, after node $i+1$ is swept, the last $\ell - i$ rows and columns of W are null. Recalling (7) and using the induction hypothesis, the matrix-product $(\Phi_i')^T W \Phi_i'$ can be written in block form as follows:

$$\left[\begin{array}{c|c|c} I & c & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & I \end{array}\right] \left[\begin{array}{c|c|c} W_{1-n..i-1,1-n..i-1} & W_{1-n..i-1,i} & 0 \\ \hline W_{i,1-n..i-1} & w_{ii} & 0 \\ \hline 0 & 0 & 0 \end{array}\right] \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline c^T & 0 & 0 \\ \hline 0 & 0 & I \end{array}\right] \begin{array}{c} 1-n \\ \vdots \\ i-1 \\ \text{row } i, \\ i+1 \\ \vdots \\ \ell \end{array}$$

which results in

$$\left[\begin{array}{c|c|c} W_{1-n..i-1,1-n..i-1} + c\, W_{i,1-n..i-1} + W_{1-n..i-1,i}\, c + w_{ii}\, c\, c^T & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \end{array}\right] \begin{array}{l} 1-n \\ \vdots \\ i-1 \\ \text{row } i. \\ i+1 \\ \vdots \\ \ell \end{array} \tag{30}$$

Thus at this point the last $\ell - (i-1)$ rows and columns have been zeroed.

Again using (27), we have

$$(\overline{\mathrm{v}})^T \Phi_i'' = \bar{v}_i [\Phi_i]_i'',$$

where the nonnull entries of $[\Phi_i]_i''$ have column and row indices that correspond to predecessors of node i. Therefore, the last $\ell - (i-1)$ rows and columns of $[\Phi_i]_i''$ are also null. Hence the second and last block assignment involving W will preserve this property, which, by induction, is valid till the end of the algorithm. ∎

Using the definition of c in (8), the componentwise translation in the first block assignment involving W in Algorihm 6 is

$$\left((\Phi_i')^T W \Phi_i'\right)_{jk} = \begin{cases} w_{jk} + \dfrac{\partial \phi_i}{\partial v_k}\dfrac{\partial \phi_i}{\partial v_j} w_{ii} + \dfrac{\partial \phi_i}{\partial v_k} w_{ji} + \dfrac{\partial \phi_i}{\partial v_j} w_{ik}, & \text{if } j < i \text{ and } k < i, \\ 0, & \text{otherwise.} \end{cases} \tag{31}$$

For the second block assignment, using (27), we have that

$$\left(\bar{v}_i [\Phi_i]_i''\right)_{jk} = \begin{cases} \bar{v}_i \dfrac{\partial^2 \phi_i}{\partial v_j \partial v_k}, & \text{if } j < i \text{ and } k < i, \\ 0, & \text{otherwise.} \end{cases} \tag{32}$$

Finally, notice that, since the componentwise version of the block assignment, done as node i is swept, involves only entries with row and column indices smaller than or equal to i, one does not need to actually zero out the row and column i of W, as these entries will not be used in the following iterations.

This componentwise assignment may be still simplified using symmetry. In order to avoid unnecessary calculations with symmetric counterparts, we employ the notation $w_{\{ji\}}$ to denote both w_{ij} and w_{ji}. Notice, however, that, when $j = k$ in (31), we have

$$\left((\Phi_i')^T W \Phi_i'\right)_{jj} = w_{jj} + \left(\frac{\partial \phi_i}{\partial v_j}\right)^2 w_{ii} + \frac{\partial \phi_i}{\partial v_j} w_{ji} + \frac{\partial \phi_i}{\partial v_j} w_{ij},$$

so in the new notation we would have

$$\left((\Phi_i')^T W \Phi_i'\right)_{\{jj\}} = w_{\{jj\}} + \left(\frac{\partial \phi_i}{\partial v_j}\right)^2 w_{\{ii\}} + 2\frac{\partial \phi_i}{\partial v_j} w_{\{ji\}}.$$

The componentwise version of Algorithm 6 adopts the point of view of the node being swept. Say, for instance that node i is being swept. Consider the first block assignment

$$W \leftarrow (\Phi_i')^T W \Phi_i',$$

whose componentwise version is given in (31). Instead of focusing on updating each $w_{\{jk\}}$, $j, k < i$, at once, which would involve accessing $w_{\{ii\}}$, $w_{\{ji\}}$ and $w_{\{ik\}}$, we focus on each $w_{\{pi\}}$ at a time, and 'push' its contribution to the appropriate $w_{\{jk\}}$'s. Taking into account that the partial derivatives of ϕ_i may only be nonnull with respect to i's predecessors, these appropriate elements will be $w_{\{jp\}}$, where $j \prec i$, see the `pushing` step in Algorithm 7.

The second block assignment

$$W \leftarrow W + (\overline{\nabla}^i)^T \Phi_i'',$$

may be thought of as the creation of new contributions, that are added to appropriate entries and that will be pushed in later iterations. From its componentwise version in (32), we see that only entries of W associated with predecessors of node i may be changed in this step. The resulting componentwise version of the `edge_pushing` algorithm is Algorithm 7.

Algorithm 7: Componentwise form of `edge_pushing`.

Input: tape $\mathcal{T}$
initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{\ell-1} = 0$, $\bar{v}_\ell = 1$, $w_{\{ij\}} = 0$, $1 - n \le j \le i \le \ell$
for $i = \ell, \ldots, 1$ **do**
  `Pushing`
  foreach *p such that* $p \le i$ *and* $w_{\{pi\}} \neq 0$ **do**
    if $p \neq i$ **then**
      foreach $j \prec i$ **do**
        if $j = p$ **then**
          $$w_{\{pp\}} += 2\frac{\partial \phi_i}{\partial v_p} w_{\{pi\}}$$
        else
          $$w_{\{jp\}} += \frac{\partial \phi_i}{\partial v_j} w_{\{pi\}}$$
        end
      end
    else $p = i$
      foreach *unordered pair* $\{j, k\}$ *such that* $j, k \prec i$ **do**
        $$w_{\{jk\}} += \frac{\partial \phi_i}{\partial v_k}\frac{\partial \phi_i}{\partial v_j} w_{\{ii\}}$$
      end
    end
  end
  `Creating`
  foreach *unordered pair* $\{j, k\}$ *such that* $j, k \prec i$ **do**
    $$w_{\{jk\}} += \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_k \partial v_j}$$
  end
  `Adjoint`
  foreach $j \prec i$ **do**
    $$\bar{v}_j += \bar{v}_i \frac{\partial \phi_i}{\partial v_j}$$
  end
end
Output: $f'' = PWP^T$

Algorithm 7 has a very natural interpretation in terms of the graph model introduced in Section 3. The nonlinear arcs are 'created' and their weight initialized (or updated, if in fact they already exist) in the `creating` step. In graph terms, the `pushing` step performed when node i is swept actually pushes the endpoints of the nonlinear arcs incident to node i to its predecessors. The idea is that subpaths containing the nonlinear arc are replaced by shortcuts. This follows from the fact that if a path contains the nonlinear arc $\{i, p\}$, then it must also contain precisely one of the other arcs incident to node i. Figure 5 illustrates the possible subpaths and corresponding shortcuts. In cases I and III, the subpaths consist of two arcs, whereas in case III, three arcs are

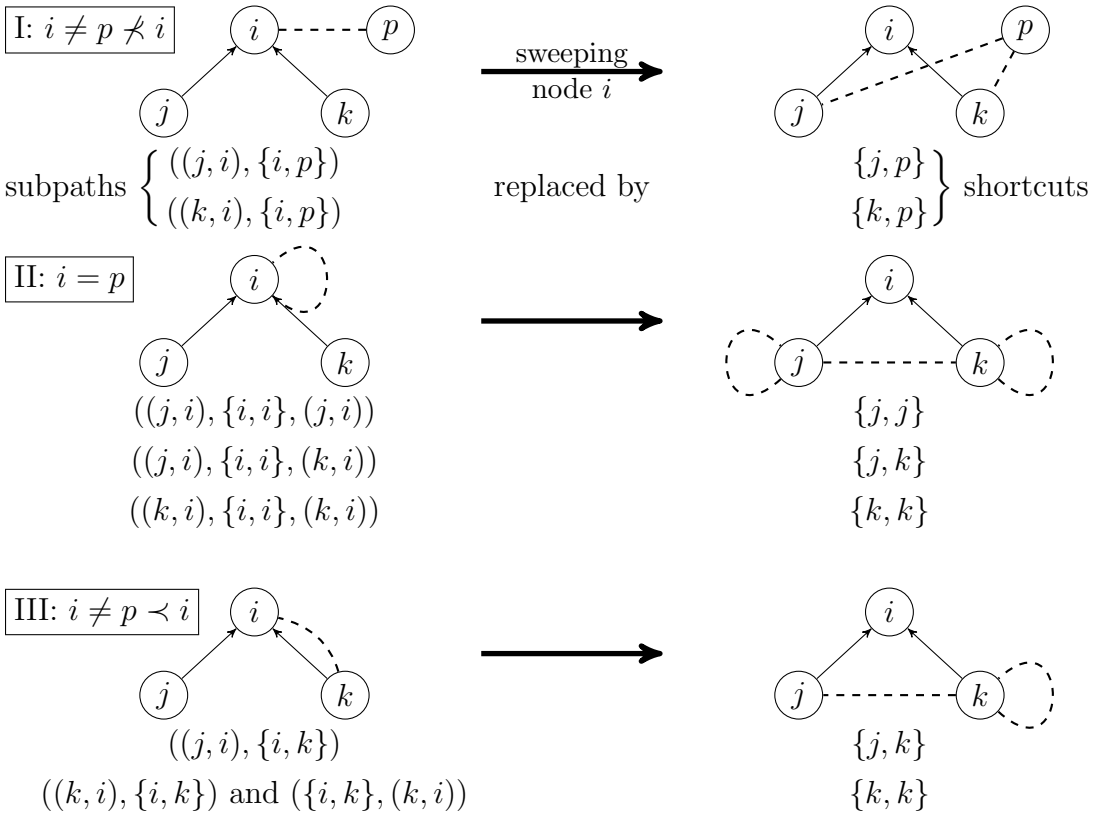


Figure 5: Pushing nonlinear arc $\{i,p\}$ is creating shortcuts.

replaced by a new nonlinear arc. Notice that the endpoints of a loop (case II) may be pushed together down the same node, or split down different nodes. In this way, the contribution of each nonlinear arcs trickles down the graph, distancing the higher numbered nodes until it finally reaches the independent nodes.

This interpretation helps in understanding the good performance of `edge_pushing` in the computational tests, in the sense that only "proven" contributions to the Hessian (nonlinear arcs) are dealt with.

6.2 Example

In this section we run Algorithm 7 on one example, to better illustrate its workings. Since we're doing it on paper, we have the luxury of doing it symbolically.

The iterations of `edge_pushing` on a computational graph of the function $f(x) = (x_{-2} + e^{x_{-1}})(3x_{-1} + x_0^2)$ are shown on Figure 6. The thick arrows indicate the sequence of three iterations. Nodes about to be swept are highlighted. As we proceed to the graph on the right of the arrow, nonlinear arcs are created (or updated), weights are appended to edges and adjoint values are updated, except for the independent nodes, since the focus is not gradient computation. For instance, when node 3 is swept, the nonlinear arc $\{1,2\}$ is created. This nonlinear arc is pushed and split into two when node 2 is swept, becoming nonlinear arcs $\{0,1\}$ and $\{-1,1\}$, with weights $1 \cdot 2v_0$ and $1 \cdot 3$, respectively. When node 1 is swept, the nonlinear arc $\{-1,1\}$ is pushed and split

into nonlinear arcs $\{-2,-1\}$ and $\{-1,-1\}$, the latter with weight $2\cdot 3\cdot e^{v_{-1}}$. Later on, in the same iteration, the nonlinear contribution of node 1, $\partial^2\phi_1/\partial v_{-1}^2$, is added to the nonlinear arc $\{-1,-1\}$. Other operations are analogous. The Hessian can be retrieved from the weights of the nonlinear arcs between independent nodes at the end of the algorithm:

$$f''(x) = \begin{pmatrix} 0 & 3 & 2v_0 \\ 3 & e^{v_{-1}}(6+v_2) & 2v_0e^{v_{-1}} \\ 2v_0 & 2v_0e^{v_{-1}} & 2v_1 \end{pmatrix} = \begin{pmatrix} 0 & 3 & 2x_0 \\ 3 & e^{x_{-1}}(6+3x_{-1}+x_0^2) & 2x_0e^{x_{-1}} \\ 2x_0 & 2x_0e^{x_{-1}} & 2(x_{-2}+e^{x_{-1}}) \end{pmatrix}.$$

Notice that arcs that are pushed are deleted from the figure just for clarity purposes, though this is not explicitly done in Algorithm 7. Nevertheless, in the actual implementation the memory locations corresponding to these arcs are indeed deleted, or, in other words, made available, since this can be done in constant time.

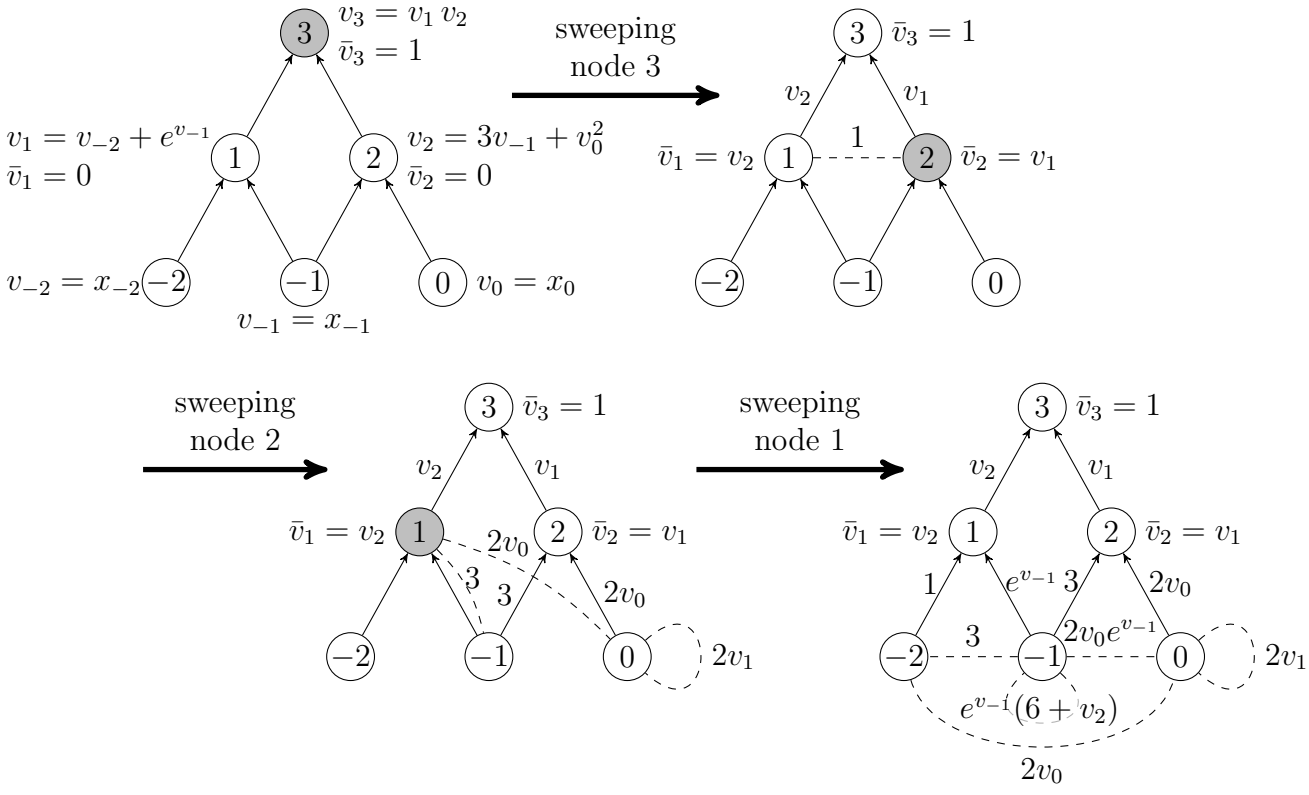


Figure 6: `edge_pushing` applied to a computational graph of $f(x) = (x_{-2} + e^{x_{-1}})(3x_{-1} + x_0^2)$.

6.3 `edge_pushing` complexity bounds

For our bounds we assume that the data structure used for W in Algorithm 7 is an adjacency list. This is a structure appropriate for large sparse graphs, which shall be our model for W, denoted by G_W. The entries in W are interpreted as the set of arc weights. Thus the nodes of G_W are associated with the rows of W. Notice that this is the same as the set of nodes of the computational graph.

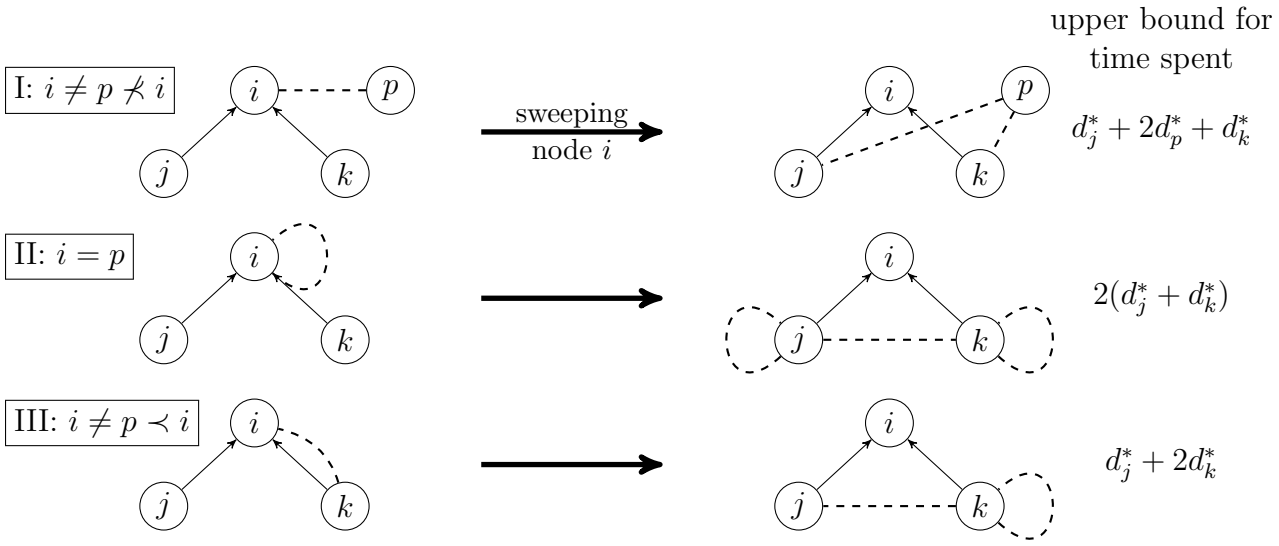


Figure 7: Complexity bounds for the pushing step.

The support of W is associated to the set of arcs of G_W. During the execution of the algorithm, new arcs may be created during the `pushing` or the `creating` step. After node i has been swept, G_W has accumulated all the nonlinear arcs that have been created or pushed, up to this iteration, since arcs are not deleted. On may think of G_W as the recorded history (creation and pushing) of the nonlinear arcs.

Denote by N_i the set of neighbors of node i in G_W and by d_i the degree of node i. Of course the degree of node i and its neighborhood vary during the execution of the algorithm. The time for inserting or finding an arc $\{i, j\}$ and its weight $w_{\{i,j\}}$ is bounded by $O(d_i + d_j)$, where d_i and d_j are the degrees at the iteration where the operation takes place. We assume that the set of elemental functions is composed of only unary and binary functions.

6.3.1 Time complexity

The time complexity of `edge_pushing` depends on how many nonlinear arcs are allocated during execution. Thus it is important to establish bounds for the number of arcs allocated to each node. Furthermore, we may fix G_W^* as the graph obtained at the end of the algorithm.

Let d_i^* be the degree of node i in G_W^*, and let $d^* = \max_i\{d_i^*\}$. Clearly $d_i \leq d_i^*$, where d_i is the degree of node i in the graph G_W at any given iteration. In order to bound the complexity of `edge_pushing`, we consider the `pushing` and `creating` steps separately. We repeat in Figure 7 the possible cases of pushing, and the corresponding time complexity bounds.

Studying the cases spelled out in Figure 7, one concludes that the time spent in pushing edge $\{i, p\}$ is bounded by $2(d_j^* + d_p^* + d_k^*)$, where j and k are predecessors of node i. Since there are at most d_i^* nonlinear arcs incident to node i, the time spent in the `pushing` step at the iteration where node i is swept is bounded by

$$d_i^*(2(d_j^* + d_p^* + d_k^*)) = O(d_i^*(d_j^* + d_p^* + d_k^*)) = O(d_i^* d^*).$$

Finally, the assumption that all functions are either unary or binary implies that at most three nonlinear arcs are allocated during the `creating` step, for each iteration of `edge_pushing`. Hence the time used up in this step at the iteration where node i is swept is bounded by

$$2(d_j^* + d_k^*) = O(d_j^* + d_k^*) = O(d^*),$$

where j and k are predecessors of node i.

Thus, taking into account the time spent in merely visiting a node — say, when the intermediate function associated with the node is linear — is constant, the time complexity of `edge_pushing` is

$$\begin{aligned} \text{TIME}(\texttt{edge\_pushing}) &\leq \sum_{i=1}^{\ell} (d_i^* d^* + d^* + 1) \\ &= O\left(d^* \sum_{i=1}^{\ell} d_i^* + \ell \right). \end{aligned} \tag{33}$$

A consequence of this bound is that, if f is linear, the complexity of `edge_pushing` is that of the function evaluation, a desirable property for Hessian algorithms.

7 Computational experiments

All tests were run on the 32-bit operating system Ubuntu 9.10, processor Intel 2.8 GHz, and 4 GB of RAM. All algorithms were coded in C and C++. The algorithm `edge_pushing` has been implemented as a driver of ADOL-C, and uses the taping and operator overloading functions of ADOL-C [8]. The tests aim to establish a comparison between `edge_pushing` and two algorithms, available as drivers of ADOL-C v. 2.1, that constitute a well established reference in the field. These algorithms incorporate the graph coloring routines of the software package *ColPack* [5, 6] and the sparsity detection and Hessian-vector product procedures of ADOL-C [14]. We shall denote them by the name of the coloring scheme employed: Star and Acyclic. Analytical properties of these algorithms, as well as numerical experiments with them, have been reported in [4, 14].

We have hand-picked fifteen functions from the CUTE collection [1] and one — augmlagn — from [9] for the experiments. The selection was based on the following criteria: Hessian's sparsity pattern, scalability and sparsity. We wanted to cover a variety of patterns; to be able to freely change the scale of the function, so as to appraise the performance of the algorithms as the dimension grows; and we wanted to work with sparse matrices. The appendix presents results for dimension values n in the set 5 000, 20 000, 50 000 and 100 000, but the tables in this section always refer to the $n = 50\,000$ case, unless otherwise explicitly noted.

The list of functions is presented in Table 1. The 'Pattern' column indicates the type of sparsity pattern: bandwith (B x), arrow, box, or irregular pattern. The last two display the number of columns of the *seed matrix* produced by Star and Acyclic, for dimension equal to 50 000. In order to report the performance of these algorithms, we briefly recall their *modus operandi*. Their first step, executed only once, computes a seed matrix S via coloring methods, such that the Hessian f'' may be recovered from the product $f''S$, which involves as many Hessian-vector products as the number of columns of S. The latter coincides with the number of colors used in the coloring of a graph model of the Hessian. The recovery of the Hessian boils down to the solution of a linear

system. Thus the first computation of the Hessian takes necessarily longer, because it comprises two steps, where the first one involves the coloring, and the second one deals with the calculation of the actual numerical entries. In subsequent Hessian computations, only the second step is executed, resulting in a shorter run. It should be noted that the number colors is practically insensitive to changes in the dimension of the function in the examples considered, with the exception of the functions with irregular patterns, noncvxu2 and ncvxbqp1.

Name	Pattern	# colors	
		Star	Acyclic
cosine	B 1	3	2
chainwoo	B 2	3	3
bc4	B 1	3	2
cragglevy	B 1	3	2
pspdoc	B 2	5	3
scon1dls	B 2	5	3
morebv	B 2	5	3
augmlagn	5×5 diagonal blocks	5	5
lminsurf	B 5	11	6
brybnd	B 5	13	7
arwhead	arrow	2	2
nondquar	arrow + B 1	4	3
sinquad	frame + diagonal	3	3
bdqrtic	arrow + B 3	8	5
noncvxu2	irregular	12	7
ncvxbqp1	irregular	12	7

Table 1: Test functions

Table 2 reports the times taken by `edge_pushing` and by the first and second Hessian computations by Star and Acyclic. It should be pointed out that Acyclic failed to recover the Hessian of ncvxbqp1, the last function in the table. In the examples where `edge_pushing` is faster than the second run of Star (resp., Acyclic), we can immediately conclude that `edge_pushing` is more efficient for that function, at that prescribed dimension. This was the case in 14 (resp., 16) examples. However, when the second run is faster than `edge_pushing`, the corresponding coloring method may eventually win, if the Hessians are computed a sufficient number of times, so as to compensate the initial time investment. This of course depends on the context in which the Hessian is used, say in a nonlinear optimization code. Thus the number of evaluations of Hessians is linked to the number of iterations of the code. The minimum time per example is highlighted in Table 2.

Focusing on the two-stage Hessian methods, we see that Star always has fastest second runtimes. Only for function sinquad is Star's first run faster than Acyclic's. Nevertheless, this higher investment in the first run is soon paid off, except for functions arwhead, nondquar and bdqrtic, where it would require over 1600, 50 and 25, respectively, computations of the Hessian to compensate the slower first run. We can also see from Tables 1 and 2 that Star's performance on the second run suffers the higher the number of colors needed to color the Hessian's graph model, which is to be expected. Thus the second runs of lminsurf, brybnd, bdqrtic, noncvxu2 and ncvxbqp1 were the slowest

	Star		Acyclic		
Name	1st	2nd	1st	2nd	e_p
cosine	9.93	0.16	9.68	2.52	0.15
chainwoo	35.07	0.33	33.24	5.08	0.30
bc4	10.02	0.25	10.00	2.56	0.25
cragglevy	28.17	0.79	28.15	2.60	0.48
pspdoc	10.31	0.35	10.27	4.39	0.23
scon1dls	11.00	0.59	10.97	4.96	0.40
morebv	10.36	0.46	10.33	4.49	0.35
augmlagn	15.99	0.68	8.36	16.74	0.27
lminsurf	9.30	1.01	9.24	3.89	0.35
brybnd	11.87	2.44	11.73	12.63	1.68
arwhead	176.50	0.16	45.86	0.24	0.20
nondquar	166.59	0.18	28.64	2.57	0.12
sinquad	606.72	0.26	888.57	1.51	0.32
bdqrtic	262.64	1.34	96.87	7.80	0.80
noncvxu2	29.69	1.10	29.27	7.76	0.28
ncvxbqp1	13.51	2.42	–	–	0.37
Averages	87.98	0.78	82.08	5.32	0.41
Variances	25 083.44	0.54	50 313.10	19.32	0.14

Table 2: Runtimes in seconds for Star, Acyclic and `edge_pushing`.

of Star's. Notice that, although the Hessian of bdqrtic doesn't require as many colors as the other four just mentioned, the function evaluation itself takes longer.

On a contrasting note, `edge_pushing` execution is not tied to sparsity patterns and thus this algorithm proved to be more robust, depending more on the density and number of nonlinear functions involved in the calculation. In fact, this is confirmed by looking at the variance of the runtimes for the three algorithms, see the last row of Table 2. Notice that `edge_pushing` has the smallest variance. Furthermore, although Star was slightly faster than `edge_pushing` in the second run for the functions arwhead and sinquad, the time spent in the first run was such that it would require over 4 000 and 10 000, respectively, evaluations of the Hessian to compensate for the slower first run.

The bar chart in Figure 8, built from the data in Table 2, permits a graphical comparison of the performances of Star and `edge_pushing`. Times for function brybnd deviate sharply from the remaining ones, it was a challenge for both methods. On the other hand, function ncvxbq1 presented difficulties to Star, but not to `edge_pushing`.

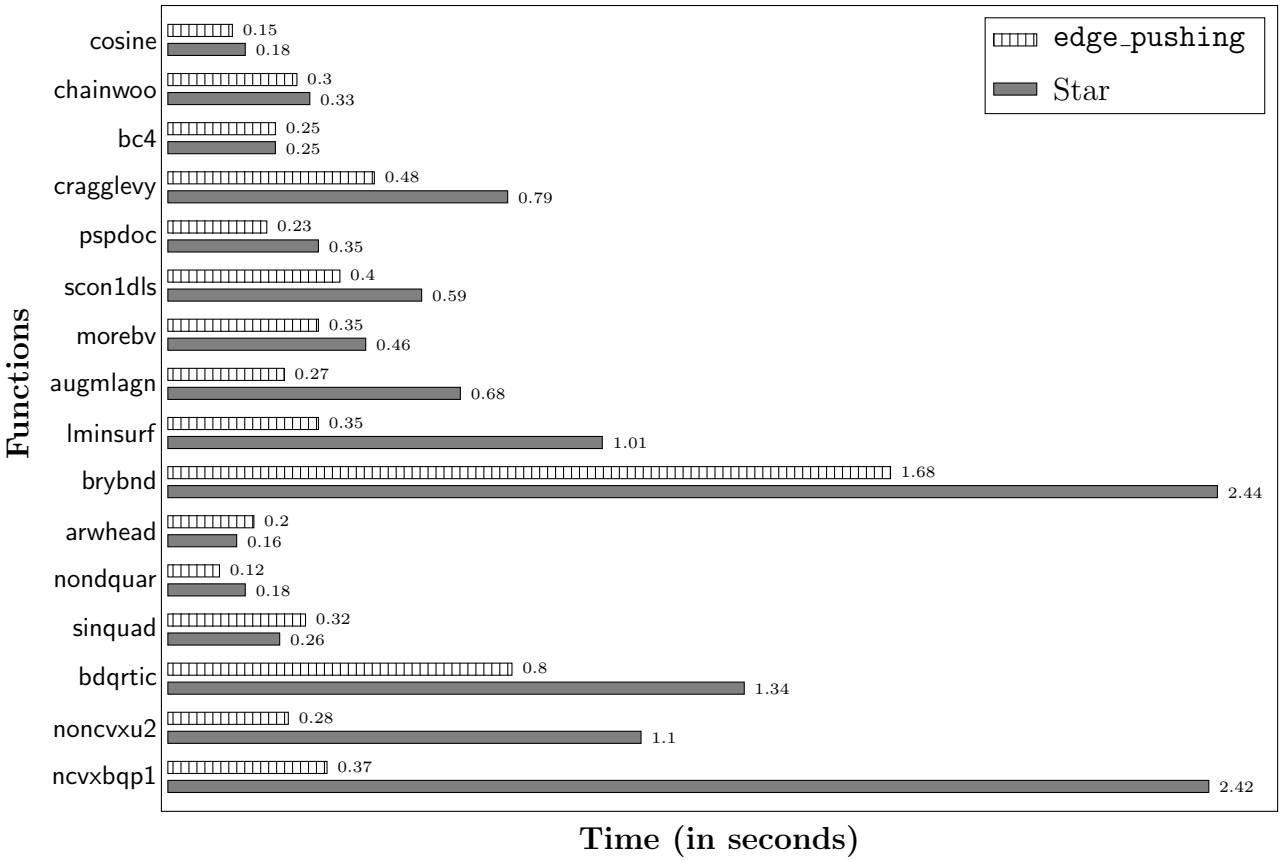


Figure 8: Graphical comparison: Star versus `edge_pushing`.

The bar chart containing the runtimes of the three algorihtms is made pointless by the range of runtimes of Acyclic, much bigger than the other two. To circumvent this problem, we applied the base 10 log to the runtimes multiplied by 10 (just to make all logs positive). The resulting chart is depicted in Figure 9.

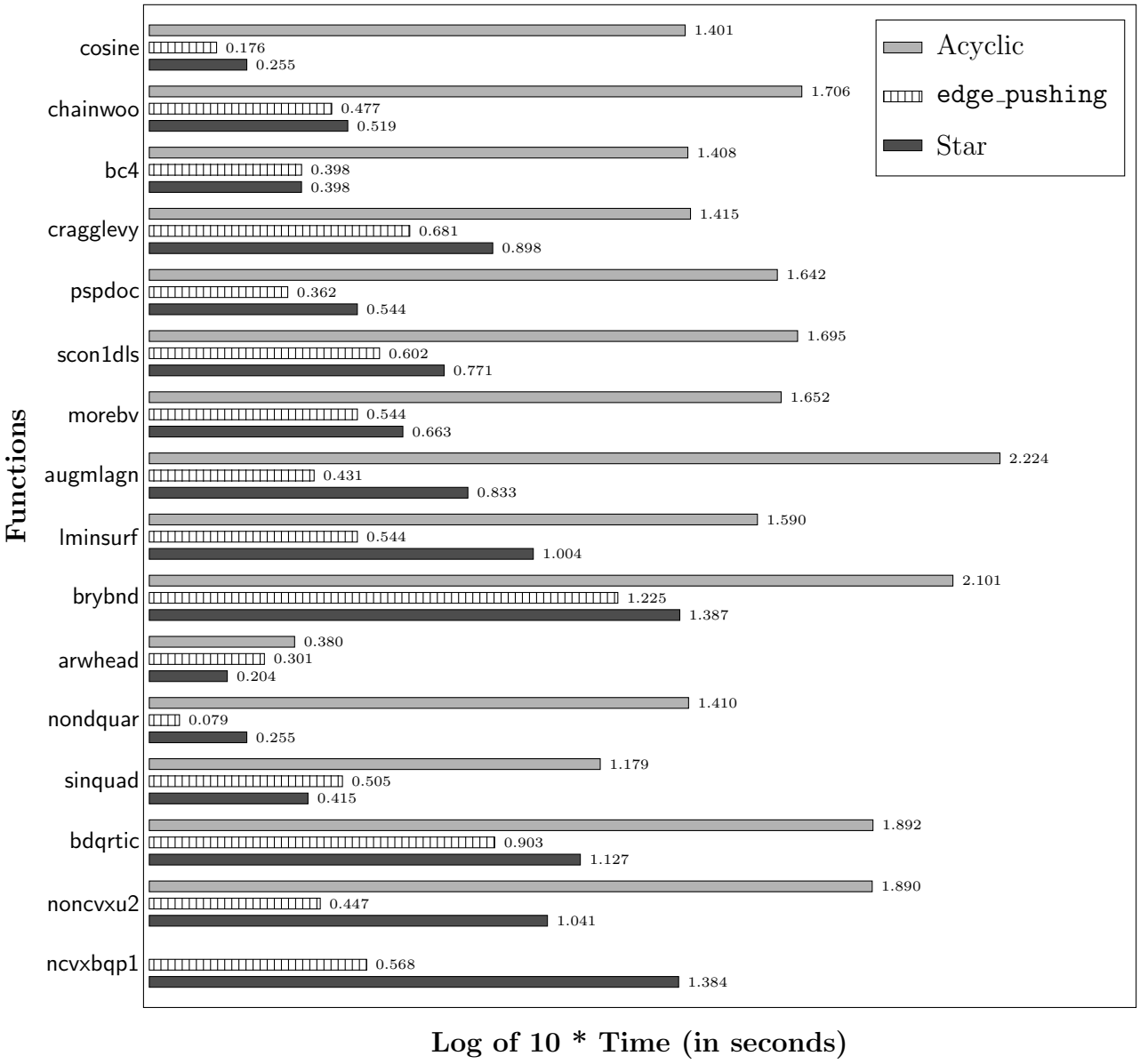


Figure 9: Graphical comparison of times in log scale: Star, Acyclic and `edge_pushing`.

Although the results presented in Table 2 correspond to the dimension 50 000 case, they represented the general behavior of the algorithms in this set of functions. This is evidenced by the plots in Figures 10 and 11, that show the runtimes of `edge_pushing` and Star on four functions for dimensions varying from 5 000 to 100 000.

The functions cosine, sinquad, brybnd and noncvxu2 were selected for these plots because they exemplify the different phenomena we observed in the 50 000 case. For instance, the performances of both `edge_pushing` and Star are similar in the functions cosine and sinequad, and this has happened consistently in all dimensions. Thus the dashed and solid lines in Figure 10 intertwine, and there is no striking dominance of one algorithm over the other. Also, these functions presented no real challenges, and the runtimes in all dimensions are low.

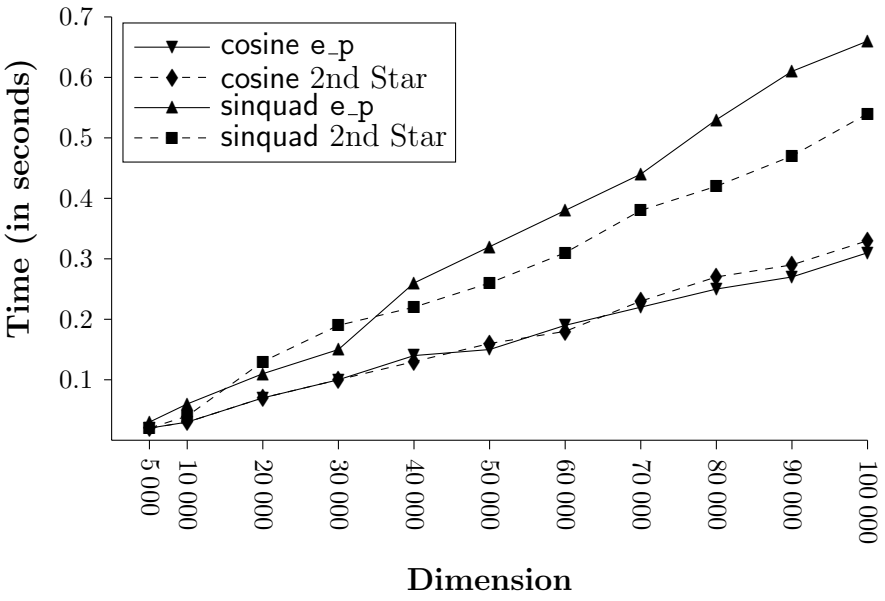


Figure 10: Evolution of runtimes of `edge_pushing` and Star (2nd run) with respect to dimension, for cosine and sinequad.

The function brybnd was chosen because it presented a challenge to all methods, and ncvxu2 is the representative of the functions with irregular Hessians. The plots in Figure 11 show a consistent superiority of `edge_pushing` over Star for these two functions. All plots are close to linear, with the exception of the runtimes of Star for the function noncvxu2. We observed that the number of colors used to color the graph model of its Hessian varied quite a bit, from 6 to 21. This highest number occured precisely for the dimension 70 000, the most dissonant point in the series.

Table 3 in the Appendix contains the runtimes for the three methods, including first and second runs, for all functions, for dimensions 5 000, 20 000 and 100 000.

8 Conclusions and future research

The formula (25) for the Hessian obtained in Section 4 leads to new correctness proofs for existing Hessian computation algorithms and to the development of new ones. We also provided a graph model for the Hessian computation and both points of view inspired the construction of

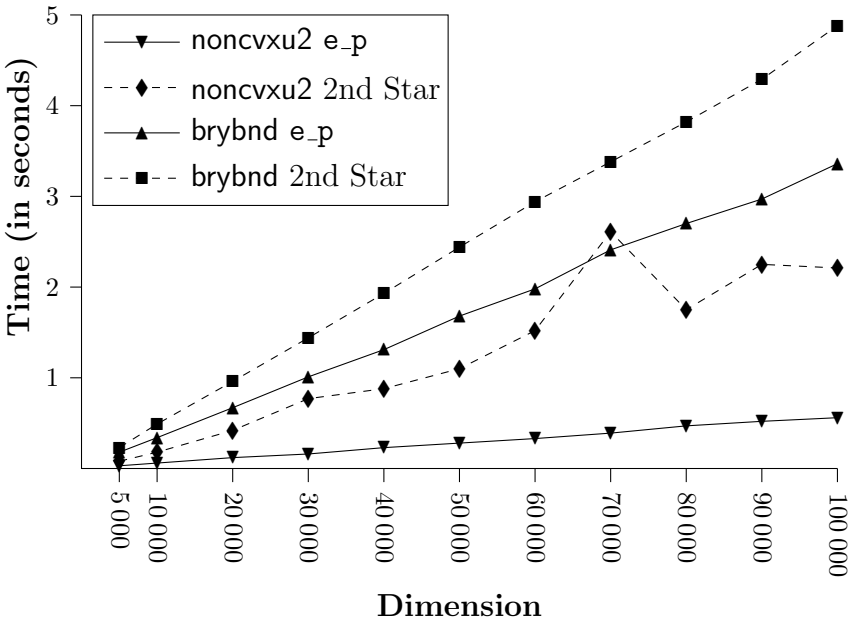


Figure 11: Evolution of runtimes of `edge_pushing` and Star (2nd run) with respect to dimension, for noncvxu2 and brybnd.

`edge_pushing`, a new algorithm for Hessian computation that conforms to Griewank and Walther's Rule 16 of Automatic Differentiation [7, p. x]:

> *The calculation of gradients by nonincremental reverse makes the corresponding computational graph symmetric, a property that should be exploited and maintained in accumulating Hessians.*

The new method is a truly reverse algorithm that exploits the symmetry and sparsity of the Hessian. It is a one-phase algorithm, in the sense that there is no preparatory run where a sparsity pattern needs to be calculated that will be reused in all subsequent iterations. This can be an advantage if the function involves many intermediate functions whose second derivatives are zero in a sizable region, for instance $h(u) = (\max\{-u, 0\})^2$. This type of function is used as a differentiable penalization of the negative axis. It is not uncommon to observe the 'thinning out' of Hessians over the course of nonlinear optimization, as the iterations converge to an optimum, which obviously lies in the feasible region. If the sparsity structure is fixed at the beginning, one cannot take advantaged of this slimming down of the Hessian.

`edge_pushing` was implemented as a driver of ADOL-C[8] and tested against two other algorithms, the Star and Acyclic methods of ColPack [6], also available as drivers of ADOL-C. Computational experiments were run on sixteen functions of the CUTE collection [1]. The results show the strong promise of the new algorithm. When compared to Star, there is a clear advantage of `edge_pushing` in fourteen out of the sixteen functions. In the remaining two the situation is unclear, since Star is a two-stage method and the first run can be very expensive. So even if its second run is faster than `edge_pushing`'s, one should take into account how many evaluations are needed in order to compensate the first run. The answers regarding the functions arwhead and sinquad were over 4 000 and 10 000, respectively, for dimension equal to 50 000. These numbers grow with the

dimension. Finally, it shoud be noted that edge_pushing's performance was the more robust, and it wasn't affected by the lack of regularity in the Hessian's pattern.

We observed that Star was consistently better than Acyclic in all computational experiments. However, Gebremedhin et al. [4] point out that Acyclic was better than Star in randomly generated Hessians and the real-world power transmission problem reported therein, while the opposite was true for large scale banded Hessians. It is therefore mandatory to test `edge_pushing` not only on real-world functions, but also within the context of a real optimization problem. Only then can one get a true sense of the impact of using different algorithms for Hessian computation.

It should be pointed out that the structure of `edge_pushing` naturally lends itself to parallelization, a task already underway. The opposite seems to be true for Star and Acyclic. The more efficient the first run is, the less colors, or columns of the seed matrix one has, and only the task of calculating the Hessian-vector products corresponding to $f''S$ can be seen to be easily parallelizable.

Another straightforward consequence of `edge_pushing` is a sparsity pattern detection algorithm. This has already been implemented and tested, and will be the subject of another report.

Appendix: Results for varying instance sizes

Dimension	5 000					20 000					100 000				
Name	Star		Acyclic		e_p	Star		Acyclic		e_p	Star		Acyclic		e_p
	1st	2nd	1st	2nd		1st	2nd	1st	2nd		1st	2nd	1st	2nd	
cosine	0.10	0.02	0.09	0.04	0.02	1.58	0.07	1.61	0.45	0.07	37	0.35	37	9.48	0.31
chainwoo	0.38	0.04	0.33	0.09	0.02	6.11	0.12	5.41	0.92	0.11	137	0.65	130	19.54	0.58
bc4	0.11	0.02	0.10	0.05	0.02	1.59	0.09	1.58	0.48	0.10	37	0.51	37	9.57	0.50
cragglevy	0.29	0.05	0.28	0.05	0.04	4.54	0.30	4.53	0.49	0.19	109	1.57	109	9.66	1.00
pspdoc	0.11	0.04	0.11	0.07	0.02	1.61	0.14	1.60	0.86	0.09	36	0.70	36	17.49	0.44
scon1dls	0.11	0.04	0.12	0.07	0.04	1.63	0.24	1.61	0.92	0.16	37	0.95	37	20.05	0.81
morebv	0.12	0.05	0.12	0.08	0.04	1.63	0.19	1.61	0.91	0.14	37	0.88	37	18.13	0.73
augmlagn	0.13	0.07	0.11	0.21	0.02	1.64	0.28	1.36	2.83	0.12	84	1.40	33	65.98	0.55
lminsurf	0.12	0.09	0.12	0.09	0.03	1.57	0.45	1.55	0.78	0.14	36	2.30	36	15.04	0.68
brybnd	0.17	0.23	0.16	0.22	0.18	1.96	0.96	1.88	2.20	0.67	39	4.88	39	42.05	3.36
arwhead	1.52	0.01	0.42	0.03	0.02	28.80	0.06	9.99	0.09	0.09	943	0.31	233	0.47	0.42
nondquar	1.29	0.01	0.21	0.04	0.01	23.19	0.08	3.49	0.48	0.05	1012	0.35	340	9.62	0.25
sinquad	2.79	0.02	5.09	0.05	0.03	60.97	0.11	99.54	0.33	0.13	3905	0.54	8961	5.14	0.66
bdqrtic	1.55	0.13	0.48	0.22	0.09	28.62	0.55	7.66	1.40	0.34	4323	2.68	833	71.4	1.65
noncvxu2	0.32	0.08	0.32	0.12	0.03	4.85	0.42	4.73	1.41	0.12	118	2.21	117	29.45	0.56
ncvxbqp1	0.15	0.20	–	–	0.02	2.22	0.91	–	–	0.13	51	5.39	–	–	0.77
Averages	0.58	0.07	0.54	0.10	0.04	10.78	0.31	9.88	0.97	0.17	684	1.60	734	22.9	0.83

Table 3: Runtimes for all methods and functions, at varying dimensions.

References

[1] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. "CUTE: constrained and unconstrained testing environment". In: *ACM Trans. Math. Softw.* 21.1 (1995), pp. 123–160. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/200979.201043. URL: http://portal.acm.org/ft_gateway.cfm?id=201043&type=pdf&coll=Portal&dl=GUIDE&CFID=106302864&CFTOKEN=87967305.

[2] R. H. Byrd, J. N., and R. A. Waltz. "KNITRO: An integrated package for nonlinear optimization". In: *Large Scale Nonlinear Optimization, 35–59, 2006*. Springer Verlag, 2006, pp. 35–59.

[3] Anders Forsgren, Philip E. Gill, and Margaret H. Wright. "Interior methods for nonlinear optimization". In: *SIAM Review* 44 (2002), pp. 525–597.

[4] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther. "Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation". In: *INFORMS J. on Computing* 21.2 (2009), pp. 209–223. ISSN: 1526-5528. DOI: http://dx.doi.org/10.1287/ijoc.1080.0286.

[5] Assefaw H. Gebremedhin, Arijit Tarafdar, Duc Nguyen, and Alex Pothen. *ColPack*. 2010. URL: http://www.cs.odu.edu/%7Ednguyen/dox/colpack/html/.

[6] Assefaw H. Gebremedhin, Arijit Tarafdar, and Alex Pothen. "COLPACK: A graph coloring package for supporting sparse derivative matrix computation". In preparation. 2008.

[7] A. Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-451-6.

[8] A. Griewank et al. *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*. Tech. rep. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167. Institute of Scientific Computing, Technical University Dresden, 1999.

[9] W. Hock and K. Schittkowski. "Test examples for nonlinear programming codes". In: *Journal of Optimization Theory and Applications* 30.1 (1980), pp. 127–129.

[10] R. H. F. Jackson and G. P. McCormick. "The polyadic structure of factorable function tensors with applications to high-order minimization techniques". In: *J. Optim. Theory Appl.* 51.1 (1986), pp. 63–94. ISSN: 0022-3239. DOI: http://dx.doi.org/10.1007/BF00938603.

[11] James Stewart. *Multivariable Calculus*. Brooks Cole, 2007.

[12] R. J. Vanderbei and D. F. Shanno. "An Interior-Point Algorithm For Nonconvex Nonlinear Programming". In: *Computational Optimization and Applications* 13 (1997), pp. 231–252.

[13] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program.* 106.1 (2006), pp. 25–57. ISSN: 0025-5610. DOI: http://dx.doi.org/10.1007/s10107-004-0559-y.

[14] A. Walther. "Computing sparse Hessians with automatic differentiation". In: *ACM Trans. Math. Softw.* 34.1 (2008), pp. 1–15. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/1322436.1322439.