Computing the Sparsity Pattern of Hessians using Automatic Differentiation

Robert Mansel Gower*and Margarida Pinheiro Mello †

May 3, 2013

Abstract

We compare two methods that calculate the sparsity pattern of Hessian matrices using the computational framework of automatic differentiation. The first method is a forward-mode algorithm by Andrea Walther in 2008 which has been implemented as the driver called `hess_pat` in the automatic differentiation package ADOL-C. The second is `edge_push_sp`, a new reverse mode algorithm descended from the `edge_pushing` algorithm for calculating Hessians by Gower and Mello in 2012. We present complexity analysis and perform numerical tests for both algorithms. The results show that the new reverse algorithm is very promising.

1 Introduction

Obtaining the sparsity pattern of the Hessian is a necessary step in a well-known graph-coloring-based method for calculating sparse Hessians using Automatic Differentiation (AD) or Finite Differences [6, 7]. With the sparsity pattern in hand, one may also use univariate Taylor series or second order scalar methods to individually calculate each nonzero element in the Hessian [1, 2].

The first suggested strategy for efficiently calculating the sparsity pattern of Hessian matrices was reported in [5]. This article considers the case of partially separable functions, i.e., functions that are a sum of nonlinear terms, to suggest that the sparsity pattern of each term be obtained separately and then all patterns should be appropriately combined. To obtain the pattern of each Hessian, [5] proposes that one should use an adapted version of the reverse AD tool for calculating Hessian vector products [4].

In 2008 Walther [18] proposed, analyzed and presented numerical results of a new algorithm for obtaining Hessian sparsity patterns. This algorithm is essentially a forward mode AD procedure and was implemented in ADOL-C [10]. Complexity bounds and tests of its use in conjunction with ColPack, a graph coloring package for calculating derivative matrices, were reported in [6]. Recently, a new reverse mode AD Hessian algorithm `edge_pushing` was developed by Gower and Mello [8], with promising results. Tests comparing `edge_pushing` to the graph-coloring-based Hessian computation algorithms implemented as drivers in ADOL-C indicate that the latter could improve considerably given a more efficient sparsity calculation step. This motivated the derivation of a sparsity algorithm from `edge_pushing`. Both sparsity calculating algorithms

*contact: Unversity of Edinburgh gowerrobert@gmail.com

†State University of Campinas margarid@ime.unicamp.br

use the framework of automatic differentiation. This framework is best described using the computational graph point of view of a function evaluation.

We consider n-variable functions that can be represented by a computational graph $\mathcal{CG} = (G, \phi)$ with the following characteristics. The directed graph $G = (V \cup Z, E)$ is acyclic, $Z = \{1-n, \ldots, 0\}$ is the set of nodes with zero indegree, called *independent nodes*, and $V = \{1, \ldots, \ell\}$ is the set of *intermediate nodes*. We assume that for each intermediate node i there is a path from some independent node to i in G. Since G is acyclic we may assume that each arc (i, j) in E satisfies $i < j$. We denote by $S(i) = \{j \mid \exists(i, j) \in E\}$ the *set of successors of node* i and by $P(i) = \{j \mid \exists(j, i) \in E\}$ its *set of predecessors*. To each node i in G we associate the variable v_i. The independent variables are associated with the zero indegree nodes. To simplify notation, we apply a shift of $-n$ to the indices of the independent variables, so that $x = (x_{1-n}, \ldots, x_0)$, and we let $v_{i-n} \equiv x_{i-n}$, for $i = 1, \ldots, n$. The variable associated with an intermediate node is a function of its predecessors, specified by the vector ϕ: $v_i = \phi_i(v_{P(i)})$, for $i \in V$. Of course the ranges and domains of the ϕ_i's must be properly defined. With this setup, given a set of values for the independent variables $x_{1-n}, \ldots, x_0$, the values of all variables $v_{1-n}, \ldots, v_\ell$ may be calculated in a forward sweep of the graph:

$$\begin{aligned} v_{i-n} &= x_{i-n}, && \text{for } i = 1, \ldots, n \\ v_i &= \phi_i(v_{P(i)}), && \text{for } i = 1, \ldots, \ell. \end{aligned}$$

Given a node i in G, let $\overline{P}(i)$ be the set of nodes j of G such that there is a path from j to i. This set may be thought of as a type of precedence closure. Another interpretation is possible if we recall that paths naturally give rise to a partial order $\preceq$ amongst nodes of an acyclic directed graph by letting $j \preceq i$ if there is a path from j to i in the graph. Then $\overline{P}(i)$ is the set of nodes comparable and less than or equal to node i in this partial order. Of course the value of v_i depends ultimately on the values associated with independent nodes in $Z_i = \overline{P}(i) \cap Z$, called i's *index domain* in [12], i.e., there is a function u_i such that $v_i = u_i(x_{Z_i})$. Sometimes it is also convenient to consider u_i as a function of the whole vector x. The computational subgraph constituted by G_i, the subgraph of G induced by nodes in $\overline{P}(i)$ and the accompanying set of functions may be thought of as the computational graph of u_i. Notice that node i is, by construction, the maximum, with respect to $\preceq$, of the nodes in G_i, henceforth called node i's *apex-induced subgraph*. Thus a computational graph $\mathcal{CG} = (G, \phi)$ is actually a computational graph of many functions, one for each intermediate node, but usually it is constructed from a program for the evaluation of a single function $f\colon D \subset \mathbb{R}^n \to \mathbb{R}^m$. In this case, without loss of generality, G has precisely m zero outdegree nodes $i_1 < \cdots < i_m = \ell$ and $f(x) = (u_{i_1}(x), \ldots, u_{i_m}(x))$, so that $\mathcal{CG} = (G, \phi)$ is called a computational graph of f. As with many other formal definitions, it is common to relax the formality when working with the defined objects. In this case, this means identifying the computational graph with its "graph" part, failing to make explicit mention of the functions and independent variables. We will adhere to this lax use when no ambiguity derives therefrom.

Figure 1 depicts a computational graph of the function $f(x) = 5x_{-2}(x_{-1} + x_0)$. The functions associated with the nodes are listed on the right of the figure. The apex-induced subgraphs of nodes 1 and 2 have node sets $\{-2, 1\}$ and $\{-1, 0, 2\}$, respectively.

In practice, each ϕ_i is taken from a group of *elemental functions* which, together with their derivatives, are

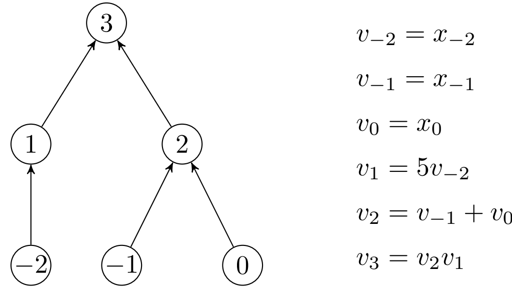

Figure 1: A computational graph of $f(x) = 5x_{-2}(x_{-1} + x_0)$.

already coded. These elemental functions typically include unary functions such as $\exp(\cdot), \sin(\cdot)$ and $\log(\cdot)$, and binary functions such as multiplication and addition. Since we are restricting ourselves to functions that may be represented by computational graphs, they are in turn compositions of elemental functions.

Once a computational graph of a function has been built, its evaluation is accomplished by doing a forward sweep of the graph, and the time spent in doing so is proportional to the number ℓ of intermediate nodes, assuming that the complexity of evaluating any elemental function is bounded above. Thus the construction of an efficient computational graph, in the sense of a small number of intermediate nodes, is of great interest. But our focus here is on the efficient computation of the sparsity pattern of the Hessian matrix of a real function f, given a (fixed) computational graph thereof. From here on we assume that f is a real valued function.

The sparsity structure that we are interested in is a structure that indicates the positions of the entries in the Hessian matrix that are not identically zero. We refer to this sparsity structure as the global sparsity pattern. Thus we distinguish between a *local* sparsity pattern, that is, the set of locations of the nonzero entries in $f''(x)$, for fixed x, and the *global sparsity* pattern, which is the set of locations (j, k) of entries in f'' such that

$$\frac{\partial^2 f}{\partial x_j \partial x_k} \not\equiv 0.$$

If f is twice continuously differentiable, the global sparsity structure of f'' is symmetric, so we refer to the global sparsity pattern of f'' as a set of unordered pairs.

An important concept for the computation of the global sparsity pattern is that of nonlinear interactions. Node j has a *nonlinear interaction* with k if there exist i, r and t such that $\partial^2 \phi_i / \partial v_r \partial v_t \not\equiv 0$ and j (resp., k) belongs to r's (resp., t's) apex-induced subgraph. Thus, for instance, nodes 1 and 2 of the graph in Figure 1 have a nonlinear interaction since $\partial^2 \phi_3 / \partial v_1 \partial v_2 \equiv 1$. This means the pairs $\{0, 1\}$, $\{-1, 1\}$, $\{-2, 2\}$, $\{-2, -1\}$ and $\{-2, 0\}$ also have nonlinear interactions, which were, we may say, inherited from the nonlinear interaction between nodes 1 and 2. These nonlinear interactions are indicated by dashed lines in Figure 2.

The relevance of nonlinear interactions is spelled out in the next proposition. Roughly speaking, the nonlinear interactions will determine the global sparsity pattern.

Proposition 1.1 *If there does not exist a nonlinear interaction between nodes j and k, then $\partial^2 u_i(x) / \partial x_j \partial x_k \equiv 0$, for all $i \in \{1, \ldots, \ell\}$.*

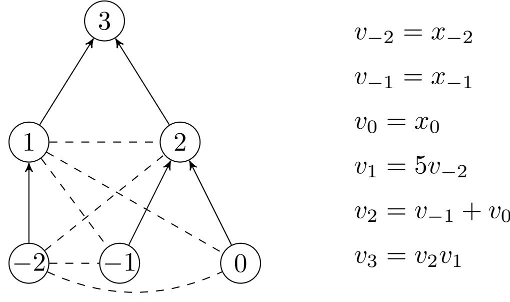

Figure 2: Dashed lines represent the nonlinear interactions between nodes of graph in Figure 1.

Proof: The proof is by induction on i. It is trivially true for $u_1(x)$ since the only predecessors of node 1 are independent nodes, and given that there is no nonlinear interaction between j and k, then $\partial^2 u_1(x)/\partial x_j \partial x_k \equiv \partial^2 \phi_1/\partial v_j \partial v_k \equiv 0$. Suppose the induction hypothesis is true for indices smaller than i. By the chain-rule we have

$$\begin{aligned}\frac{\partial^2 u_i}{\partial x_j \partial x_k} &= \frac{\partial}{\partial x_j}\left(\sum_{r\in P(i)} \frac{\partial \phi_i}{\partial v_r}\frac{\partial u_r}{\partial x_k}\right) \\ &= \sum_{r\in P(i)}\left(\frac{\partial u_r}{\partial x_k}\frac{\partial}{\partial x_j}\left(\frac{\partial \phi_i}{\partial v_r}\right) + \frac{\partial \phi_i}{\partial v_r}\frac{\partial^2 u_r}{\partial x_j \partial x_k}\right).\end{aligned}$$

By remembering that the partial derivative $\partial\phi_i/\partial v_r$ is evaluated at $v_{P(i)} = u_{P(i)}(x)$, we may apply the chain-rule once again to obtain

$$\frac{\partial^2 u_i}{\partial x_j \partial x_k} = \sum_{r,t\in P(i)} \frac{\partial u_r}{\partial x_k}\frac{\partial^2 \phi_i}{\partial v_t \partial v_r}\frac{\partial u_t}{\partial x_j} + \sum_{r\in P(i)} \frac{\partial \phi_i}{\partial v_r}\frac{\partial^2 u_r}{\partial x_j \partial x_k}. \tag{1}$$

The node numbering scheme assumed implies $r < i$, thus the induction hypothesis is true for node r, and the rightmost term in (1) is zero. The leftmost term on the right-hand side of (1) is also zero, for, if it were not, there would exist a nonlinear interaction between node j and k. ∎

Applying the proposition to $u_\ell(x) = f(x)$, we conclude that, if $\partial^2 f/\partial x_j \partial x_k \not\equiv 0$, then there exists a nonlinear interaction between nodes j and k. Therefore, one can build an overestimate of the global sparsity pattern by including all nonlinear interaction pairs. This may be an overestimate due to degeneracy. For example consider the function f defined by the following sequence of statements

$$v_0 = x_0,\ v_1 = 1/v_0,\ v_2 = v_1 v_0.$$

It easily follows that $v_2 \equiv 1$, hence $f'' \equiv 0$ and its global sparsity pattern should be empty, but the overestimate would include the pair $\{0, 0\}$. Nevertheless, for most computational graphs, this overestimated global sparsity pattern coincides with the global sparsity pattern. From here on we drop the "overestimated" adjective for brevity.

2 Forward mode algorithm: `hess_pat`

Roughly speaking, the `hess_pat` algorithm sweeps forward through a function's computational graph and, upon encountering a nonlinear elemental function ϕ_i, checks to see how ϕ_i contributes to the global sparsity pattern of the Hessian. To do this, it calculates and stores all index domains. The description below is adapted from [18], taking into account that here all variable indices have been shifted by $-n$. By initially setting $Z_i = \{i\}$, for $i = 1-n, \ldots, 0$, the remaining index domains are computed using the forward recurrence:

$$Z_i \leftarrow \bigcup_{t \in P(i)} Z_t, \quad \text{for } i = 1, \ldots, \ell.$$

To keep track of the accumulating global sparsity pattern, Walther [18] defines the *nonlinear interaction domains* $\mathcal{N}_j$, for $j = 1-n, \ldots, 0$, as follows:

$$\mathcal{N}_j := \{k \leq 0 \, : \, j \text{ has a nonlinear interaction with } k\}.$$

The pseudo code of `hess_pat` is in Algorithm 2.1. The algorithm performs a forward sweep of the intermediate nodes from 1 to ℓ. As node i is being swept, Z_i is calculated by merging the index domains of i's predecessors. Then, if ϕ_i is a nonlinear function, for each ordered pair (k,t) such that $\partial^2\phi_i/\partial v_k \partial v_t \not\equiv 0$, one knows that every node in k's apex-induced subgraph, including the independent variables indexed in Z_k, has a nonlinear interaction with each node in t's apex-induced subgraph, which includes the independent variables indexed in Z_t. Thus the nonlinear interaction domain $\mathcal{N}_p$, for each p in Z_k, is updated by merging Z_t into $\mathcal{N}_p$. If k is not equal to t, then the pair (t,k) will also be examined, and $\mathcal{N}_p$, for each $p \in Z_t$, will also be updated accordingly. At the end, the nonlinear interaction domains $\mathcal{N}_j$, for $j = 1-n, \ldots, 0$, contain the global sparsity information.

Algorithm 2.1: Forward mode: `hess_pat`

Input: A computational graph $G = (V \cup Z, E)$ of $f(x)$.
Initialization: $Z_i = \{i\}$, $\mathcal{N}_i = \emptyset$, for $i = 1-n, \ldots, 0$
for $i = 1, \ldots \ell$ **do**
  $Z_i \leftarrow \bigcup_{t \in P(i)} Z_t$
  if ϕ_i *is nonlinear* **then**
    foreach $(k,t) \in P(i) \times P(i)$ *such that* $\dfrac{\partial^2 \phi_i}{\partial v_k \partial v_t} \not\equiv 0$ **do**
      foreach $p \in Z_k$ **do**
        $\mathcal{N}_p \leftarrow \mathcal{N}_p \cup Z_t$
Output: The nonlinear interaction domains $\mathcal{N}_j$, for $j = 1-n, \ldots, 0$.

3 A new reverse mode algorithm: `edge_push_sp`

In Algorithm 3.1, `edge_push_sp`, we suggest a novel and, in many practical instances, more efficient way of obtaining the global sparsity pattern, that accumulates the nonlinear interactions in a reverse sweep of the

computational graph. In `edge_push_sp`, a dynamic undirected graph[1] $H = (V \cup Z, W)$, with the same node set as the computational graph, is used to store the nonlinear interactions between nodes. For each node i, let N_i be the set of i's neighbors in H. Thus $t \in N_i$ if and only if $\{t, i\} \in W$. Also, given that H is an undirected graph, $t \in N_i$ if and only if $i \in N_t$.

A nonlinear interaction between two variables is represented by an undirected edge linking their respective nodes in H. At the beginning W is empty (thus the neighborhood sets are empty). A nonlinear interaction between nodes r and t may stem from the nonlinearity of the elemental function associated with a common successor of r and t, or be inherited. Accordingly, in the `Creating` step, when node i is being swept, we will add edges between predecessors r and t of node i whenever $\partial^2\phi_i/\partial v_r \partial v_t$ is not identically zero. (Notice that we may have $r = t$.) Then, during the `Pushing` step, the nonlinear interactions of the node being swept are "pushed down" to its predecessors. In this way, the nonlinear information is propagated backwards with respect to the node order, until, at the end of the algorithm we have the nonlinear interactions between nodes in Z, and thus know the global sparsity pattern of f''.

`Pushing` is accomplished in three steps. The first considers the existence of a loop edge incident to node i, i.e., $i \in N_i$. In this case, each predecessor $t \in P(i)$ has a nonlinear interaction with each predecessor $k \in P(i)$. Notice that, since we do not assume any special behavior for the neighbor list data structure, to add to H an undirected edge between k and t we first insert t in N_k, when the ordered pair (k, t) is examined, and then add k to N_t, when the ordered pair (t, k) shows up in the `foreach` loop. [2] This includes the case that $k = t$, so whenever there is a loop incident to a node, all its predecessors will have loops as well. The loop incident to node i, if existent, is deleted at the end of this first step. The second step in `Pushing`, which is a loop in t, merges N_i into N_t, for each $t \in P(i)$, which effectively pushes down i's remaining nonlinear interactions to its predecessors. The third step and loop then adds the mirror images of the edges created in the previous step, to maintain the undirected characteristic of the edges. We also remove the edge $\{j, i\}$ for each $j \in N_i$ in this loop and at the end of the `Pushing` operation, we empty N_i, for, ultimately, we are not interested in nonlinear interactions between intermediate nodes.

At the end of the algorithm, N_j coincides with the nonlinear interaction domain $\mathcal{N}_j$, for $j = 1 - n, \ldots, 0$, and thus the set of edges of H contains the global sparsity pattern.

This sparsity calculating algorithm is descended from `edge_pushing`, a reverse Hessian AD algorithm [8]. The added suffix "`_sp`" stands for sparsity pattern. Contrary to what's done in `hess_pat`, the contribution of a nonlinear function ϕ_i to the global sparsity pattern is not immediately calculated. Instead, the occurrence of a nonlinear function initiates a trickle of edges down the computational graph. Only when these edges reach the independent nodes is their contribution to the sparsity pattern known. It should be noted that, unlike `edge_pushing`, which calculates the actual entries of the Hessian, `edge_push_sp` does not require a forward sweep of the computational graph.

The workings of the algorithm are illustrated in Figure 3, which graphically shows the iterations of

[1] Strictly speaking, H is a multigraph, given that loops are permitted. Since the allowance of loops will be clear throughout the article, we adopt the shorter term.

[2] This is done to assist in the complexity analysis, for in a real implementation, ones front end to a data structure would omit this symmetric operation.

Algorithm 3.1: Reverse mode: `edge_push_sp`.

Input: A computational graph $G = (V \cup Z, E)$ of $f(x)$.

Initialization: Undirected graph H, with node set $V \cup Z$ and neighborhood sets $N_j = \emptyset$, for $j = 1-n, \ldots, \ell$.

for $i = \ell : 1$ **do**

  `Creating:`

  if ϕ_i *is nonlinear* **then**

    foreach $(k,t) \in P(i) \times P(i)$ *such that* $\dfrac{\partial^2 \phi_i}{\partial v_k \partial v_t} \not\equiv 0$ **do**

      $N_k \leftarrow N_k \cup \{t\}$

  `Pushing:`

  if $i \in N_i$ **then**

    foreach $k \in P(i)$ **do**

      $N_k \leftarrow N_k \cup P(i)$

    $N_i \leftarrow N_i \setminus \{i\}$

  foreach $t \in P(i)$ **do**

    $N_t \leftarrow N_t \cup N_i$

  foreach $j \in N_i$ **do**

    $N_j \leftarrow N_j \setminus \{i\}$

    $N_j \leftarrow N_j \cup P(i)$

  `Empty`(N_i)

Output: The sparsity pattern of f's Hessian, represented by the sets N_j, for $j = 1-n, \ldots, 0$.

`edge_push_sp` on a computational graph of the function $f(x) = 3x_{-2}\exp(x_{-1} + x_0)$. The thick arrows indicate the sequence of four iterations. Nodes about to be swept are highlighted. As we proceed to the graph on the right of the arrow, nonlinear arcs are created and pushed. Starting with the sweeping of node 4, the nonlinear arc $\{3, 2\}$, represented by a undirected dashed edge, is created for $\partial^2\phi_4/\partial v_3 \partial v_2 \not\equiv 0$. In the subsequent iteration, this nonlinear arc is pushed to node 3's only predecessor, node -2. As node 2 is swept, the edge $\{1, 1\}$ is created, for $\partial^2\phi_2/\partial v_1^2 \not\equiv 0$. After this, the edge $\{2, -2\}$ is pushed, resulting in the new edge $\{1, -2\}$. In the final iteration, the loop $\{1, 1\}$ is pushed and consequently the edges $\{-1, -1\}$, $\{-1, 0\}$ and $\{0, 0\}$ are produced. Finally, edge $\{1, -2\}$ is pushed, producing edges $\{-1, -2\}$ and $\{0, -2\}$. The output indicates that the global sparsity structure contains all pairs of independent nodes, with the exception of the loop incident on node -2. The corresponding changes effected on the neighbor lists are shown in Table 1.

4 `hess_pat` Bounds for Special Partially Separable Functions

As defined in [15], a function $f : \mathbb{R}^n \to \mathbb{R}$ is partially separable if

$$f(x) = \sum_{r=1}^{m} f_r(x_{I_r}), \tag{2}$$

where $I_r \subset \{1, \ldots, n\}$, $\cup_{r=1}^{m} I_r = \{1, \ldots, n\}$, and there exists $p \in \mathbb{N}$ such that: $|I_r| \le p < n$, for $r = 1, \ldots, m$. Additionally, we assume that we have a computational graph for this partially separable function and that

	Neighbors after sweeping node			
	4	3	2	1
N_{-2}		2	1	−1, 0
N_{-1}				−2, −1, 0
N_0				−2, −1, 0
N_1			−2, 1	
N_2	3	−2		
N_3	2			
N_4				

Table 1: Evolution of neighbor lists when `edge_pushing` is applied to a computational graph of $f(x) = 3x_{-2}\exp(x_{-1} + x_0)$.

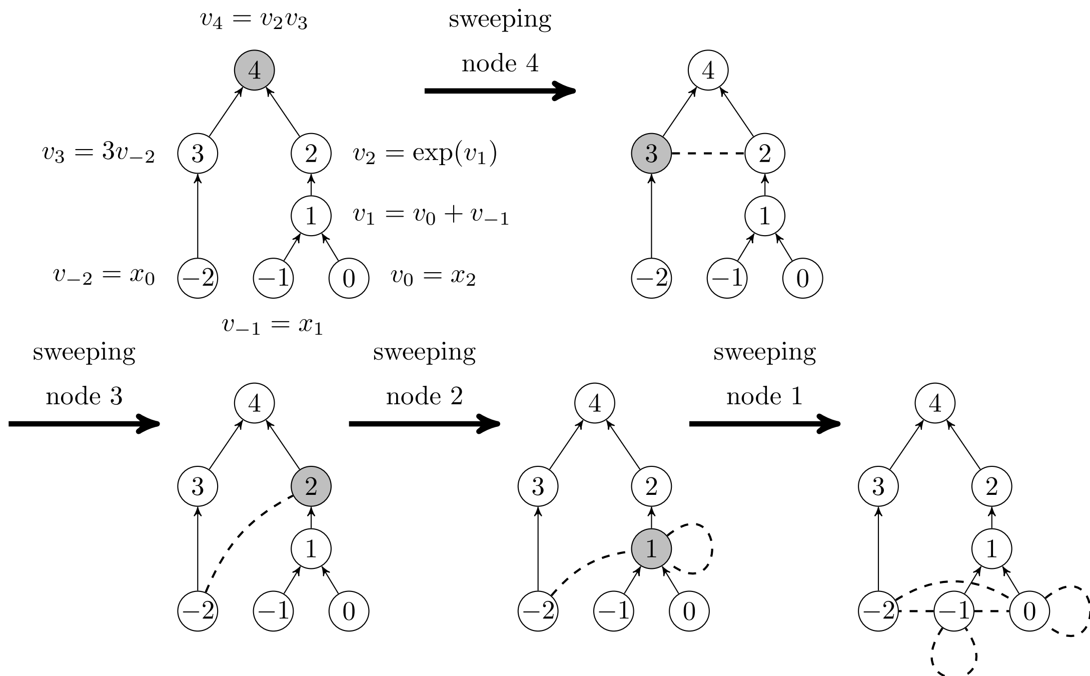

Figure 3: `edge_pushing` applied to a computational graph of $f(x) = 3x_{-2}\exp(x_{-1} + x_0)$.

the number of intermediate variables used to calculate $f_r(x_{I_r})$ is bounded above by a fixed integer q, for $r = 1, \ldots, m$. When $p \ll n$, this representation can be exploited in a number of contexts such as nonlinear optimization [15] and efficient calculation of derivatives [9].

Typically, when a coded routine of a partially separable function uses a single internal variable, say `SUM`, to accumulate the partial sums (3), the computational graph associated usually turns out to be a *PASETAG* (PArtially SEparable TAll Graph). This has certainly been the case when the graph is generated by the operator-overloading-based routines of ADOL-C. Each node f_r in Figure 4 is the apex of a subgraph that calculates the nonlinear function $f_r(x_{I_r})$. These apex-induced subgraphs are not shown in the picture to keep it simple, but one must bear in mind that they might intersect. The s_j nodes are the partial sums:

$$s_j = \sum_{r=1}^{j+1} f_r(x_{I_r}), \ \ j = 1, \ldots, m-1. \tag{3}$$

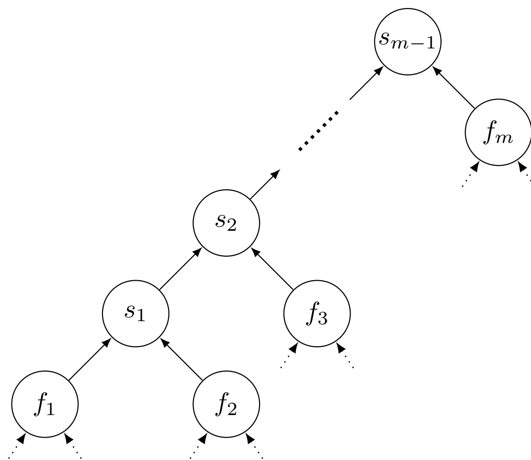

Figure 4: Partially separable tall graph.

For the bounds developed in this section, we adopt the usual assumption that each node has at most two predecessors. Furthermore, we use the same merging algorithm for sets as in [18], where the number of operations required to merge two sparse array structures is two times the cardinality of the first set plus the cardinality of the second set. Given a suitable choice of data structure, and allowing repetitions of elements in the data set, it is possible to merge sets in $O(1)$ [17]. Notice however that allowing repetitions may lead to appreciable increases in allocated memory.

The notation $f(x) = \Omega(g(x))$ means that $f(x)$ is asymptotically bounded below by the function $g(x)$ times a fixed constant. We denote by $OPS(A \leftarrow A \cup B)$ the number of operations needed to merge the set B into A. Thus, by our assumption in the previous paragraph, $OPS(A \leftarrow A \cup B) = 2|A| + |B|$.

Proposition 4.1 *The number of operations needed in the version of* `hess_pat`*, Algorithm 2.1 above, implemented in ADOL-C-2.3.0, to merge all index domains of a PASETAG is* $\Omega(n^2/p + m)$.

Proof: Let Z_{s_j} be the index domain of the partial sum node s_j, $j = 1, \ldots, m-1$. Let Z_r be the index domain of node f_r, $r = 1, \ldots, m$. We will obtain a lower bound for calculating the index domains by bounding the

complexity of computing each Z_{s_j}, $j = 1, \ldots, m-1$. The set Z_{s_1} is built by merging Z_1 and Z_2, while Z_{s_j} is built by merging the sets $Z_{s_{j-1}}$ and Z_{j+1}, for $j = 2, \ldots, m-1$. Necessarily $|Z_{s_{m-1}}| = n \leq |Z_{s_{m-2}}| + |Z_m|$, hence $OPS(Z_{s_{m-1}} \leftarrow Z_{s_{m-2}} \cup Z_m) \geq n$, for two times $|Z_{s_{m-2}}|$ plus $|Z_{s_m}|$ is at least n.

Given there are at most p independent variables in the apex-induced subgraph of f_m, we have that $|Z_{s_{m-2}}| \geq n - p$ which in turn implies that $OPS(Z_{s_{m-2}} \leftarrow Z_{s_{m-3}} \cup Z_{m-1}) \geq (n-p)$. By induction we have that $OPS(Z_{s_{m-j}} \leftarrow Z_{s_{m-j-1}} \cup Z_{m-j+1}) \geq \max\{1, n-(j-1)p\}$.

Thus, the number of operation in calculating all $Z_{s_{m-j}}$ sets, for $j = 1, \ldots, m-1$, is

$$\begin{aligned}
\sum_{j=1}^{m-1} OPS(Z_{s_{m-j}} \leftarrow Z_{s_{m-j-1}} \cup Z_{m-j+1}) &\geq n + (n-p) + \cdots + \left(n - \left\lfloor \frac{n}{p} \right\rfloor p\right) + \underbrace{1 + \cdots + 1}_{m-2-\lfloor n/p \rfloor} \\
&= \frac{1}{2}\left(2n - \left\lfloor \frac{n}{p} \right\rfloor p\right)\left(\left\lfloor \frac{n}{p} \right\rfloor + 1\right) + \left(m - 2 - \left\lfloor \frac{n}{p} \right\rfloor\right) \\
&\geq \frac{n}{2}\left(\left\lfloor \frac{n}{p} \right\rfloor + 1\right) + \left(m - 2 - \left\lfloor \frac{n}{p} \right\rfloor\right) \\
&= \frac{n-2}{2}\left(\left\lfloor \frac{n}{p} \right\rfloor + 1\right) + (m-1).
\end{aligned}$$

On the last inequality we used $n - \left\lfloor \frac{n}{p} \right\rfloor p \geq 0$. Hence the operation count for calculating the index domains in `hess_pat` is bounded below by a constant times $n^2/p + m$, commonly abbreviated as $\Omega(n^2/p + m)$. ■

Notice that the computation of the index domains accounts for only part of the computational effort. Thus, if p remains unchanged as n grows, `hess_pat`'s runtime will grow at least quadratically in n. Indeed, this is the behavior observed in the computational experiments reported in Section 6.

5 Bounds for `edge_push_sp`

To analyze the number of operations required by Algorithm 3.1 `edge_push_sp`, let us assume that the data structure used to represent the graph H is an array of adjacency lists, so each node corresponds to an element of the array and its neighboring set is a linked list. The number of operations in sweeping through a list of neighbors, to insert a new node and to delete a list of neighbors, is bounded by the size of this list. Furthermore, `edge_push_sp` uses the same merging procedure as `hess_pat`, thus has an operation count equal to two times the cardinality of the set that "receives" the merger plus the cardinality of the second set.

For this bound we extend the definition of nonlinear interaction domain to every node $t \in V \cup Z$,

$$\mathcal{N}_t = \{j \in V \cup Z \,|\, j \text{ has a nonlinear interaction with } t\}.$$

Let $\hat{n} = \max_t\{|\mathcal{N}_t|\}$.

Proposition 5.1 *The number of operations required by* `edge_push_sp` *on a given computational graph is* $O\left(\ell + \hat{n}\sum_{i=1}^{\ell} |\mathcal{N}_i|\right)$.

Proof:

Statement	Upper bound on # operations
for $i = \ell : 1$ **do**	
`Creating:`	
if ϕ_i *is nonlinear* **then**	
foreach $(k,t) \in P(i) \times P(i)$ *such that* $\dfrac{\partial^2 \phi_i}{\partial v_k \partial v_t} \not\equiv 0$ **do**	
$N_k \leftarrow N_k \cup \{t\}$	$\sum_{k \in P(i)} (2\|N_k^i\| + 1)$
(Note: at this point, the number elements in k's neighbor list is at most $\|N_k^i\| + 2$, for $k \in P(i)$.)	
`Pushing:`	
if $i \in N_i$ **then**	
foreach $k \in P(i)$ **do**	
$N_k \leftarrow N_k \cup P(i)$	$\sum_{k \in P(i)} (2(\|N_k^i\| + 2) + 2)$
$N_i \leftarrow N_i \setminus \{i\}$	$\|N_i^i\|$
foreach $k \in P(i)$ **do**	
$N_k \leftarrow N_k \cup N_i$	$\sum_{k \in P(i)} (2(\|N_k^i\| + 2) + \|N_i^i\|)$
foreach $j \in N_i$ **do**	
$N_j \leftarrow N_j \setminus \{i\}$	$\sum_{j \in N_i^i} (\|N_j^i\| + 2)$
$N_j \leftarrow N_j \cup P(i)$	$\sum_{j \in N_i^i} (2(\|N_j^i\| + 2) + 2)$
`Empty`(N_i)	$\|N_i^i\|$

Figure 5: Upper bounds on the number of operations required by the statements of Algorithm 3.1

The number of operations of `edge_push_sp` is intimately related to the number of neighbors of each node in H. Therein lies the difficulty in calculating a bound, for H is a dynamic graph, something easy to overlook since the notation does not explicitly account for this temporal dependency. Although the node set of H is fixed, the arc set may vary from one iteration to the next. To emphasize this evolution, we let N_t^i be node t's linked list of neighbors at the beginning of the iteration where node i is swept.

Let us analyze the number of operations in each iteration of `edge_push_sp`. Consider the effort of sweeping a node i. To help understand the development, we summarize in Figure 5 the upper bounds on the number of operations needed in the various calculations of `edge_push_sp` on a generic computational graph. On the left we have the several loops and statements and on the right the upper bound on the (total) number of operations needed for executing a command (inside a given loop).

The `Creating` step may add, at most, $P(i)$ elements in N_k^i, for each $k \in P(i)$. Thus at most two elements are included in N_k^i. The number of operations in the first inclusion is $|N_k^i|$ and in the second one is $|N_k^i| + 1$. Therefore the loop in the `Creating` step uses up at most $\sum_{k \in P(i)} (2|N_k^i| + 1)$ operations.

The possibly larger value of the cardinality of the neighbor's lists of each of the predecessors of i is taken into account when estimating the number of operations needed in the mergers in the three loops in the

`Pushing` step. This overestimate is also used in the third loop, since node $j \in N_i$ may also be a predecessor of node i. The elimination of i from N_i^i uses at most $|N_i^i|$ operations.

Summing up the upper bounds on the number of operations needed when sweeping node i we have

$$\#\text{operations when sweeping node } i \leq \sum_{k\in P(i)} (6|N_k^i| + 11 + |N_i^i|) + \sum_{j\in N_i^i} (3|N_j^i| + 8) + |N_i^i|. \tag{4}$$

To arrive at our final result, it remains to show that, for fixed $i, t \in V \cup Z$, we have that $N_t^i \subset \mathcal{N}_t$İf an edge $\{j, t\}$ is allocated in the creating step, then there exists k such that $\partial^2\phi_k/\partial v_t \partial v_j \not\equiv 0$, and there is a nonlinear interaction between them. Otherwise, this edge was allocated in the pushing step. Let $\overline{S}(j)$ be the set of nodes i in the computational graph G such that there is a path from j to i. Thus $\overline{S}(j)$ is the transitive closure of the successor relation. Then, through simple induction, there exists $s \in \overline{S}(j)$ and $r \in \overline{S}(t)$ such that the edge $\{s, r\}$ was allocated in the creating step, fulfilling the definition of nonlinear interaction. Thus $N_t^i \subset \mathcal{N}_t$ and, consequentially,$|N_t^i| \leq |\mathcal{N}_t| \leq \hat{n}$. Using this combined with (4), we see that the number of operations needed when sweeping node i is bounded by

$$\begin{aligned}
\sum_{k\in P(i)} (6|N_k^i| + 11 + |N_i^i|) + \sum_{j\in N_i^i} (3|N_j^i| + 8) + |N_i^i| \quad &\leq \quad \sum_{k\in P(i)} (6\hat{n} + 11 + |\mathcal{N}_i|) + \sum_{j\in N_i^i} (3\hat{n} + 8) + |\mathcal{N}_i| \\
&\leq \quad 2(6\hat{n} + 11 + |\mathcal{N}_i|) + |\mathcal{N}_i|(3\hat{n} + 8) + |\mathcal{N}_i| \\
&\leq \quad O\left(\hat{n}|\mathcal{N}_i| + 1\right).
\end{aligned}$$

Finally, the total effort of sweeping all ℓ nodes in the graph is $O\left(\ell + \sum_{i=1}^{\ell} \hat{n}|\mathcal{N}_i|\right)$. ∎

Though this is a rather coarse bound, it tells us that `edge_push_sp` ultimately only depends on nonlinear aspects of the computational graph. For instance, a linear function will have a computational graph that has no nonlinear elemental functions, thus there are no nonlinear interactions, and the number of operations carried out by `edge_push_sp` will be $O(\ell)$. This is different for `hess_pat` whose runtime is dominated by calculations involving linear interactions.

We now move on to compare the two algorithms, `edge_push_sp` and `hess_pat`, through computational tests where the focus is time taken in execution.

In terms of memory usage and complexity, most reverse AD algorithms must store the intermediate values v_i, for $i = 1 - n, \ldots, \ell$ in the forward sweep, so that the elemental functions can be evaluated in the reverse sweep. This can be debilitating for functions with extremely large computational graphs, for it may be impossible to simultaneously store all intermediate variables, even in terms of sequential access memory. A proposed solution to this problem is creating checkpointing schedules, in which memory requirements are reduced in exchange for execution time [11]. The algorithm `edge_push_sp` is free of such problems, for the elemental functions are not evaluated. Instead, the global sparsity pattern is constructed by simply inspecting whether the elemental function associated with each node is linear or nonlinear.

6 Computational Experiments

All tests were run on the 64-bit operating system Fedora Scientific, processor Intel Pentium 4 CPU 3.20GHz, and 3 GB of RAM. All algorithms were coded in C and C++. Both `edge_push_sp` and `hess_pat` algorithms

name	Pattern	hess_pat	edge_push_sp
binary	sparse B 1	60	78
cosine	B 1	3420	164
bc4	B 1	3467	314
cragglevy	B 1	15103	321
chainwoo	B 2	25131	314
pspdoc	B 2	3220	187
scon1dls	B 2	3212	365
morebv	B 2	4365	391
augmlagn	5 × 5 diagonal blocks	5666	404
lminsurf	B 5	3766	363
brybnd	B 5	3977	1120
bdexp	B 5	3769	251
chainros_trigexp	B 3 + D 6	3478	366
toiqmerg	B 7	3308	470
arwhead	arrow	22359	304
nondquar	arrow + B 1	7440	142
sinquad	frame + diagonal	17756	290
bdqrtic	arrow + B 3	31032	582
noncvxu2	irregular	26671	334
ncvxbqp1	irregular	5964	185
ncvxqp3	irregular	5446	141

Table 2: Description of problem set, Hessian sparsity pattern and execution times in milliseconds of `hess_pat` and `edge_push_sp`, for n=50'000

have been implemented as drivers of ADOL-C, and use the same taped evaluation procedure, which represents a computational graph, produced by ADOL-C [10]. For these tests, we used version ADOL-C-2.3.0, the most recent available [3]

We have hand-picked fifteen problems from the CUTE collection [3], augmlagn from [13], binary from [16], toiqmerg (Toint Quadratic Merging problem) and chainros_trigexp (Chained Rosenbrook function with Trigonometric and exponential constraints) from [14] for the experiments. Our tests consist of calculating the global sparsity structure of the Hessians of the Lagrangian functions of each problem with constraints and Hessians of the objective function otherwise. The selection was based on the following criteria: Hessian's sparsity pattern, dimension scalability and sparsity. We wanted to cover a variety of patterns; to be able to easily change the dimension of the function, so as to appraise the performance of the algorithms as it grows; and we wanted to work with sparse matrices. See Table 2. The 'Pattern' column indicates the type of sparsity pattern: bandwidth[4] of value x (B x), arrow, frame, number of diagonals (D x), or irregular pattern. The runtimes of `hess_pat` and `edge_push_sp` for $n = 50\,000$ (dimension) are in Table 2, where one can see that the `edge_push_sp` algorithm had a substantially faster runtime in all cases. Both algorithms produced precisely the same global sparsity structures.

To get a feeling for the asymptotic behavior of both algorithms, we have tabulated their runtimes on

[3] As checked on January 23, 2013.

[4] The bandwidth of matrix $M = (m_{ij})$ is the maximum value of $|i - j|$ such that $m_{ij} \neq 0$.

	morebv		sinquad		brybnd		chainros_trigexp		ncvxbqp1	
n	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp
1000	3	6	9	4	8	22	5	6	3	3
11000	158	72	786	53	216	240	175	121	212	35
21000	538	135	3059	103	674	484	565	310	718	63
31000	1173	211	7897	200	1587	859	1207	731	1540	101
41000	2214	276	13450	196	2227	1364	2226	975	3228	170
51000	3037	392	20070	296	3406	1567	3157	1616	4261	165
61000	4475	408	25519	346	4781	2152	4407	1794	6442	206
71000	5964	489	38707	371	6592	2250	5892	2594	8038	299
81000	7823	663	47174	404	8434	2190	9058	3734	10579	273
91000	9795	685	56132	548	10858	2128	10017	4104	13080	319

Table 3: Runtime results in millisecond for edge_push_sp (abbreviated to e_p_sp) and hess_pat (abbreviated to hess_p) over varying dimensions.

morebv, sinquad, brybnd, chainros_trigexp and ncvxbqp1, with varying dimension n, in Table 3. Each of these five functions is a representative of a type of sparsity pattern that was tested. Using the notation of Section 4, all test cases are partially separable, where the number of nonlinear terms m is a linear function of n, whereas p and q are independent of n, for large n.[5] Upon inspection of the computational graphs generated by ADOL-C of the coded instances of these functions, we found that many were PASETAGs, thus motivating our definition. An example that does not fit the definition of PASETAG, was the computational graph of scon1dls, which is two PASETAGs connected at the root. The reason for this was that two variables were used to store the accumulating sum of nonlinear terms.

In these circumstances, according to Proposition 4.1, hess_pat's runtime should grow, at least, quadratically. The numerical results in Table 3 corroborate this prediction. To further reveal this asymptotic behavior we have plotted the runtimes of the algorithms on brybnd, as a function of dimension, in Figure 6. All test functions exhibit similar plots, thus we have shown only the one.

For partially separable functions, accumulating linear dependencies between intermediate variables and independent variables, such as the computation of the index domains in hess_pat, becomes costly as the dimension of the problem grows. This situation can be improved if a previous step is implemented that "balances"[6] the computational graph in terms of its height. With a perfectly "balanced" graph, the lower bound on hess_pat can be reduced. Another solution, recently proposed by Walther [17], is to use a special data structure for merging the index domains that allows repetition of elements, but merges with a time complexity of $O(1)$. With such a data structure, the complexity results of Proposition 4.1 are no longer valid. In a test suit of five problems [17], this has produced a significant speed-up.

In contrast, when accumulating nonlinear dependencies in a reverse sweep, we need not carry these linear dependencies. Instead, only known contributions to the sparsity pattern are dealt with. Gebremedhin *et al.* [6] point out that the computational cost of using hess_pat to calculate Hessians became a bottleneck as the dimension n increased, in their four step graph coloring algorithms for calculating Hessians, where

[5]In a few cases, p and q grow linearly for small n, but would reach an upper bound for $n = 14$ and grow no more.

[6]Quotation marks are necessary for such a term is only defined for trees.

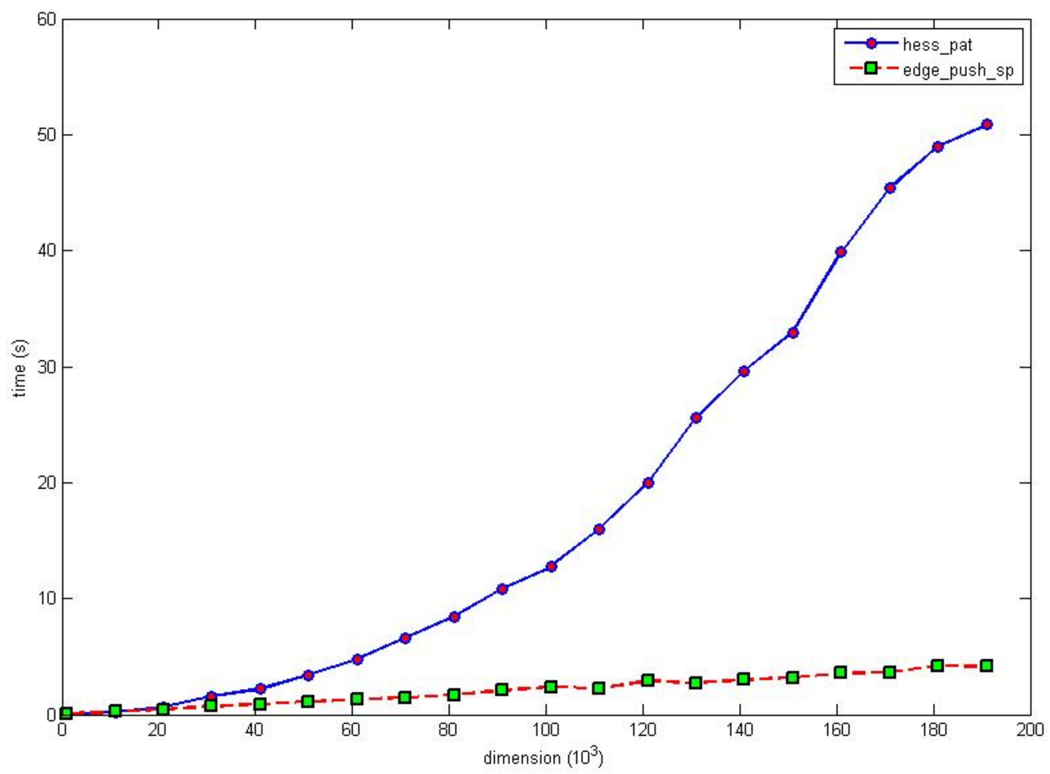

Figure 6: Runtime in seconds of both algorithms, over varying dimensions, on problem brybnd.

obtaining the sparsity pattern was the first step. This was also evidenced in [8], where tests comparing graph coloring-based approaches to `edge_pushing` revealed that the graph-coloring-based algorithms were not as competitive due to the comparatively large amount of time spent in the first two steps, comprised of calculating the sparsity pattern using `hess_pat` and graph coloring. This indicates that the efficiency of graph-coloring-based algorithms for calculating Hessians could be improved by employing `edge_push_sp` to obtain the sparsity pattern.

References

[1] Jason Abate et al. “Algorithms and design for a second-order automatic differentiation module”. In: *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*. New York: ACM, 1997, 149–155 (electronic).

[2] C. Bischof, G. Corliss, and A. Griewank. “Structured second-and higher-order derivatives through univariate Taylor series”. In: *Optimization Methods and Software* 2.3 (1993), pp. 211–232. ISSN: 1055-6788.

[3] I. Bongartz et al. “CUTE: constrained and unconstrained testing environment”. In: *ACM Trans. Math. Softw.* 21.1 (1995), pp. 123–160. ISSN: 0098-3500.

[4] Bruce Christianson. “Automatic Hessians by reverse accumulation”. In: *IMA J. Numer. Anal.* 12.2 (1992), pp. 135–150. ISSN: 0272-4979.

[5] D.M. Gay. “More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability”. In: *Computational Differentiation: Applications, Techniques, and Tools* (1996), pp. 173–184.

[6] Assefaw H. Gebremedhin et al. “Efficient computation of sparse Hessians using coloring and automatic differentiation”. In: *INFORMS J. Comput.* 21.2 (2009), pp. 209–223. ISSN: 1091-9856.

[7] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. “What color is your Jacobian? Graph coloring for computing derivatives”. In: *SIAM Rev.* 47.4 (2005), 629–705 (electronic). ISSN: 0036-1445.

[8] R. M. Gower and M. P. Mello. “A new framework for the computation of Hessians”. In: *Optimization Methods and Software* 27.2 (2012), pp. 251–273. eprint: `http://www.tandfonline.com/doi/pdf/10.1080/10556788.2011.580098`.

[9] Andreas Griewank. “Some bounds on the complexity of gradients, Jacobians, and Hessians”. In: *Complexity in numerical optimization.* World Sci. Publ., River Edge, NJ, 1993, pp. 128–162.

[10] Andreas Griewank, David Juedes, and Jean Utke. “Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++”. In: *ACM Trans. Math. Softw.* 22 (2 1996), pp. 131–167. ISSN: 0098-3500.

[11] Andreas Griewank and Andrea Walther. “Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation”. In: *ACM Trans. Math. Softw.* 26.1 (2000), pp. 19–45.

[12] Andreas Griewank and Andrea Walther. *Evaluating derivatives.* Second Edition. Principles and techniques of algorithmic differentiation. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2008, pp. xxii+438. ISBN: 978-0-898716-59-7.

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

[14] Ladislav Luksan, Jan Vlcek. *Test Problems for Unconstrained Optimization san Test Problems for Unconstrained Optimization.* Tech. rep. 897. Academy of Sciences of the Czech Republic, 2003.

[15] Ph.L. Toint and A. Griewank. “On the unconstrained optimization of partially separable objective functions”. In: ed. by M.J.D. Powell. Academic Press, London, 1982. Chap. Nonlinear Optimization 1981, pp. 301–312.

[16] E. Varnik. “Exploitation of structural sparsity in algorithmic differentiation”. PhD thesis. RWTH Aachen, 2011.

[17] A. Walther. “On the Efficient Computation of Sparsity Patterns for Hessians”. In: *Proceedings of AD 2012: Recent Advances in Algorithmic Differentiation, Lecture Notes in Computational Science and Engineering, 87*. 2012, pp. 139–149.

[18] Andrea Walther. “Computing sparse Hessians with automatic differentiation”. In: *ACM Trans. Math. Software* 34.1 (2008), Art. 3, 15. ISSN: 0098-3500.