To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–18

LLNL IM Release number: LLNL-JRNL-720198

On efficient Hessian computation using the edge pushing algorithm in Julia

C. G. Petra[a] and F. Qiang[b*] and M. Lubin[c] and J. Huchette[c]

[a]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA;* [b] *Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL;* [c]*Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA;*

(*January 2017*)

Evaluating the Hessian matrix of second-order derivatives at a sequence of points can be costly when applying second-order methods for nonlinear optimization. In this work, we discuss our experiences implementing the recently proposed Edge Pushing (EP) method in Julia as an experimental replacement for the current coloring-based methods used by JuMP, an open-source algebraic modeling language. We propose an alternative data structure for sparse Hessians to avoid the use of hash tables and analyze the space and time complexity of EP method. In our benchmarks, we find that EP is very competitive in terms of both preprocessing time and Hessian evaluation time. We identify cases where EP closes the performance gap between JuMP's previous implementation and the implementation in AMPL, a commercial software package with similar functionality.

Keywords: automatic differentiation; sparse Hessian computation; edge pushing; algebraic modeling language; Julia; JuMP

AMS Subject Classification: 90C04, 90C05, 90C06, 90C30, 65D25

1. Introduction

We focus on the problem of efficient automatic computation of sparse Hessian matrices. The edge pushing (EP) algorithm was introduced in [1] as an alternative to popular coloring-based compression algorithms [2]. Recent work by Wang et al. [3] has refined the algorithm and corrected issues with its initial implementation. However, to the best of our knowledge the EP algorithm is not yet available in any off-the-shelf automatic differentiation (AD) package.

Here we report our experiences implementing the EP algorithm within JuMP [4], an algebraic modeling language for optimization. JuMP currently implements the coloring-based algorithms for Hessian computations, and in recent benchmarks the computation time was observed to be within a factor of 2.2 of AMPL, a commercial software package with similar functionality to JuMP. Our motivation for experimenting with the EP algorithm is to potentially further close this performance gap.

Note that, in the context of optimization, we are primarily interested in the case of repeated Hessian evaluations using the same sparsity pattern. In contrast, many of the

*Corresponding author. Email: f.qiang@gmail.com

reported gains of the EP algorithm over coloring approaches are due to the expense of the coloring step, which can be amortized over many Hessian evaluations. As Wang et al. [3] state:

> For repeated Hessian computation when the sparsity pattern remains the same, the compression-based approach should be preferred over `LIVARH` or `LIVARHACC`,

where `LIVARH` and `LIVARHACC` are two variants of the EP algorithm. We would like to identify cases where the EP algorithm is superior, even, perhaps, after amortizing the cost of the coloring step.

Our initial implementation following [3] had notable performance issues due to the use of hash tables throughout the algorithm. We will describe an improved data structure we designed to implement the EP algorithm using fast $O(1)$ lookups with fewer memory indirections. This is achieved by using a storage scheme for the Hessian similar to the compressed sparse column (CSC) storage, but allowing replicated entries. We analyze the EP algorithm equipped with the proposed data structure to bound the worst-case growth in the number of replicated entries and rule out the possibility of excessive space complexity that would prevent using the EP implementation for large-scale Hessians.

This proposed data structure yielded a drastic improvement, by as much as a factor of ten, in performance over our initial implementation; it has also revealed optimization instances where the EP algorithm is indeed superior or competitive with coloring approaches. While some of the slowness of hash tables may be attributed to their current implementation in the Julia language, we believe that our new data structures could yield improvements in implementations in other languages as well.

2. Preliminaries: The edge-pushing algorithm

We consider twice continuously differentiable scalar functions $f : \mathbb{R}^n \to \mathbb{R}$ that can be represented by a finite sequence of elementary functions $\phi_1, \phi_2, \ldots, \phi_l$. The elementary functions are arithmetic operators or elementary mathematical functions, and each elementary function ϕ_i is a function of only previous functions ϕ_j defined in the sequence (i.e. $1 \leq j < i$). For example, such representations in terms of elementary functions occur naturally in the process of programming f as a computer routine. We follow the notation of Griewank and Walther [5] and denote by $v_{l-n}, v_{2-n}, \ldots, v_0$ the independent variables or the arguments of f. Furthermore, whenever ϕ_i for $i \in \{1, 2, \ldots, l\}$ is used as an argument to another elementary function, we will call it a *dependent variable* and denote it with v_i. This is needed to distinguish between the dual role of "argument" and "function" that the elementary functions ϕ_i have. Also, by $v_j \prec v_i$ we mean that v_j is an argument of ϕ_i, or that v_j *precedes* v_i, where necessarily $l - n \leq j < i \leq l$. The interdependences between the variables can be expressed by means of a directed acyclic graph, known as the computational graph (CG) of f, with the nodes representing the variables and the vertices representing the precedence relations [3].

The edge pushing (EP) algorithm introduced by Gower and Mello [1] and refined by Wang et al. [3] is an alternative to the popular coloring algorithms [2, 6–8] for efficient computation of sparse Hessians. Coloring algorithms for Hessian (H) evaluation work by computing the directional derivatives in the form of Hessian-vector products $\nabla^2 f \cdot d$, and they reduce the number of directions d needed by solving a graph coloring problem such as [2]. The graph coloring problem is NP-hard and is usually solved by specialized heuristics such as [8]. The coloring approach has two potential drawbacks: (i) the cost of the graph coloring heuristic algorithm can be unreasonably high, and/or (ii) the number

of colors needed, in either the heuristic solution or the true minimal solution, may be excessively large, leading to a large number of Hessian-vector products. In Section 4 we show instances of optimization problems for both (i) and (ii) apply.

The edge pushing algorithm takes a different approach. It applies the chain rule at each node in the computational graph and uses a Hessian graph model that augments the computational graph with new edges that track dependences of the gradients (adjoint variables) on the variables, or carry second-order partial derivatives between the variables with nonlinear relationships. This Hessian graph model can be simplified to remove symmetric redundancies [1, 3] and to derive a componentwise, recursive expression for each (k, j) Hessian entries that is not zero [3]. The recursive expression makes use of the *live variable sets* S_i $(i = \{1, \ldots, l\})$, which track the nonlinear relationships between the variables and are defined recursively by $S_{l+1} = \emptyset$ and

$$S_i = \{S_{i+1} \backslash \{i\}\} \cup \{j | v_j \prec v_i\}, \ i \in \{1, \ldots, l\}. \tag{1}$$

The live variables sets allow the derivation of a recursive expression [3] for each Hessian entry of the form

$$\forall i \in \{1, \ldots, l\}, \forall (k, j) \in S_i \times S_i,$$
$$h_{kj}^{(i)} = h_{kj}^{(i+1)} + \frac{\partial \phi_i}{\partial v_j} h_{ik}^{(i+1)} + \frac{\partial \phi_i}{\partial v_k} h_{ij}^{(i+1)} + \frac{\partial \phi_i}{\partial v_j} \frac{\partial \phi_i}{\partial v_k} h_{ii}^{(i+1)} + \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_j \partial v_k}. \tag{2}$$

Here we denote the intermediate Hessians entries by $h^{(i)} \in \mathbb{R}^{(n+l)\times(n+l)}$. The final Hessian $H \in \mathbb{R}^{(n+l)\times(n+l)}$, where entry H_{kj} is the partial second-order derivative with respect to v_j and v_k, is given by $h^{(1)}$ [3]. The adjoint variable $\bar{v}_i$ is computed during the reverse sweep computation of the gradient.

3. A variant of the Hessian edge-pushing algorithm

Numerical EP algorithms [1, 3] require frequent random access to (or lookup for) the entries of intermediate Hessians $h^{(i)}$ as required by the recursion (2). Since we target very large (but sparse) problems, the Hessian matrices need to be a stored in a data structure with low space complexity that takes advantage of the sparsity. Reducing space usually competes with the time complexity of random access, even though it does not necessarily have to do so. In this work we propose a data structure for the Hessian computation decribed by (2) similar to the compressed sparse column (CSC) storage, but allowing duplicate entries, whose sum is the value of the Hessian entry. The goal is to reduce the memory indirection overhead in the EP algorithm associated with the access to the Hessian entries in (2).

The idea of using duplicate entries comes from the observation that the Hessian recursion (2) can be manipulated to use set-valued Hessian entries. We will denote the set-valued entry (k, j) at step i in the recursion (or EP algorithm iteration) by $H_{kj}^{(i)}$. Based on the precedence relationship of k and j with respect to step i, similarily to [3],

(2) can be used to write

$$\forall i = \{l, \dots, 1\},\ \forall (j,k) \in S_i \times S_i,\ H_{jk}^{(l+1)} = \emptyset \text{ and}$$

$$H_{jk}^{(i)} = \begin{cases} H_{jk}^{(i+1)} & v_j \not\prec v_i, v_k \not\prec v_i \\ \left\{ \frac{\partial \phi_i}{\partial v_j} \cdot h : h \in H_{ik}^{(i+1)} \right\}, & v_j \prec v_i, v_k \not\prec v_i, \\ \left\{ \frac{\partial \phi_i}{\partial v_j} h : h \in H_{ik}^{(i+1)} \right\} \cup \left\{ \frac{\partial \phi_i}{\partial v_k} h : h \in H_{ij}^{(i+1)} \right\} & \\ \quad \cup \left\{ \frac{\partial \phi_i}{\partial v_j} \frac{\partial \phi_i}{\partial v_k} \cdot h : h \in H_{ii}^{(i+1)} \right\} \cup \left\{ \bar{v}_i \cdot \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \right\}, & v_j \prec v_i, v_k \prec v_i. \end{cases} \tag{3}$$

These set-valued entries $H_{jk}^{(i+1)}$ store the terms that contribute to (j,k) entry when visiting the live sets corresponding to node i. Our variant of the EP algorithm updates the set-valued entries $H_{jk}^{(i)}$ according to the recursion (3) at the benefit of $O(1)$ access time for each of the (duplicate) entry. This is discussed in detail below, in Section 3.1. When comparing our modification to the similar recursion used by Wang et al. [3] that gives $h_{jk}^{(i)}$ from (2), one can easily see that $h_{jk}^{(i)} = \sum_{h \in H_{jk}^{(i)}} h$, and, consequently, the entry (k,j) in the Hessian will be $H_{jk} = \sum_{h \in H_{jk}^{(1)}} h$. The resulting variant of the edge-pushing algorithm is very similar to the original edge-pushing algorithm [1] and is listed in Algorithm 1. In Algorithm 1 we denote by $H_{\{jk\}}$ the set containing the replications corresponding to entries (j,k) and (k,j) during the algorithm and $H_{\{jk\}}$ and $H_{\{kj\}}$ refers to the same set. Due to this "symmetric" notation, the update in (Push3) is slightly different than in (3) and it follows [1]. Also, for compactness, in Algorithm 1 we do not check whether the partials derivatives are zero; in the implementation, the entries corresponding to such partials are not appended.

3.1 *The data structure for the Hessian*

The edge-pushing algorithm described above allows us to store the intermediary Hessians using CSC storage with duplicate entries. More specifically, we use a vector of $n + l$ vectors, where the ith inner vector contains pairs (`row`, `value`) corresponding to entries on column i, possibly with multiple pairs for the same entry as required by the use of set-valued entries in (3). With this data structure, the iteration over the edges that may need to be pushed when sweeping node i—namely, the *for* in line 2 in Algorithm 1—is done by looping over the inner vector corresponding to column i, which contain all p such that $W_{\{pi\}} \neq \emptyset$. In other words, accessing $W_{\{pi\}}$ for a given p is done in $O(1)$ time. The update of $H_{\{jk\}}$ in (Push1), (Push2), (Push3), or (Create) is done by appending the corresponding pair (k, `value`) pair to the jth inner vector ($j > k$), which also has constant $O(1)$ time. This is possible because we perform an initial "preprocessing" sweep in which we compute the sizes of each inner vector and preallocate them.

In essence, accessing and updating the Hessian graph model is done in $O(1)$ when using the set-valued CSC data structure. This is a significant departure from previous edge-pushing implementations, which use either an adjacency list with access time linear in the degree of i in the Hessian graph model [1], or *std::map* which has logarithmic access complexity [3]. The downside of our approach is a potential increase in the number of elements of the set-valued Hessian entries, which results in an increased number of loops in the *for* loop in line 2, and an increase in space complexity. We discuss this trade-off in more detail in the next section.

Algorithm 1 Proposed variant of edge_pushing that uses set-valued Hessian entries

Input: tape $\mathcal{T}$
Initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{l-1} = 0, \bar{v}_l = 1, H_{\{ij\}} = \emptyset, 1-n \le j \le i \le l$

for $i = l, \ldots, 1$ **do**
  for p such that $p \le i$ and $H_{\{pi\}} \neq \emptyset$ **do**
    (*pushing step*)
    if $p \neq i$ **then**
      for $j \prec i$ **do**
        if $j = p$ **then**
          $H_{\{pp\}} = H_{\{pp\}} \cup \left\{ 2\frac{\partial \phi_i}{\partial v_p} h : h \in H_{\{pi\}} \right\}$ **(Push3)**
        else
          $H_{\{jp\}} = H_{\{jp\}} \cup \left\{ \frac{\partial \phi_i}{\partial v_j} h : h \in H_{\{pi\}} \right\}$ **(Push1)**
        end if
      end for
    else $p = i$
      for each unordered pair $\{j, k\}$ such that $j, k \prec i$ **do**
        $H_{\{jk\}} = H_{\{jk\}} \cup \left\{ \frac{\partial \phi_i}{\partial v_k} \frac{\partial \phi_i}{\partial v_j} h : h \in H_{\{ii\}} \right\}$ **(Push2)**
      end for
    end if
  end for
  (*creating step*)
  for each unordered pair $\{j, k\}$, such that $j, k \prec i$ **do**
    $H_{\{jk\}} = H_{\{jk\}} \cup \left\{ \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_k \partial v_j} \right\}$ **(Create)**
  end for
  (*adjoint step*)
  for $j \prec i$ **do**
    $\bar{v}_j += \bar{v}_i \frac{\partial \phi_i}{\partial v_j}$
  end for
end for

An alternative approach for a Julia implementation could use vectors of dictionaries (one dictionary for each row), which can theoretically provide $O(1)$ access time without needing to replicate Hessian entries. We have tried this approach and discovered that performs considerably worse than the CSC data structure, very likely because of the overhead associated with hashing.

At the end of the algorithm, the entries corresponding to the independent variables are summed to obtain the Hessian. The complexity of this final step is linear in the number of entries corresponding to the independent variables when the accumulation is done in a separate buffer, usually provided by the optimization solver.

3.2 *Complexity analysis*

Allowing for repeated Hessian entries is a considerable departure from previous implementations of the edge pushing algorithm, and can potentially limit real-world applications of the algorithm for large problems because of high space and memory requirements. We first discuss the space complexity, namely the growth in cardinality of the set-valued Hessian entries. The "push" and "create" operations in the algorithm depend on the

computational graph and the nonlinearity of the dependent nodes with respect to their predecessors in the computational graph. Nonlinearity refers to whether the partials in the four updates of Algorithm 1 are nonzero. We will work under the very conservative assumption that all the dependent variables are nonlinear with respect to their predecessors. As a result, our worst-case estimate of the space complexity (and time complexity) holds for rather isolated, "corner" cases.

We derive the the maximum growth in the cardinality of the (j,k) entry during Algorithm 1 for any given pair of nodes j and k. For this we consider four cases, depending on whether j and k are dependent of independent nodes: (i) $j,k \geq 1$, (ii) $j \geq 1$ and $k < 1$, (iii) $j < 1$ and $k \geq 1$, and (iv) $j,k < 1$. In what follows, we denote by $c_{jk}^{(i)}$ the cardinality of $H_{jk}^{(i)}$ and by $s(j)$ the number of successors $i \succ j$ of j. For compactness we use $j \prec i$ for $v_j \prec v_i$ and also use $i \succ j$ and $j \prec i$ interchangeably.

Case (i): $j,k \geq 1$**.** We first remark that the subgraph of the expression's computation graph that correspond to the dependent nodes is a tree, *i.e.*, any dependent node has only one successor (or parent). Also note that, according to (3), $H_{jk}^{(i)}$ can only be updated when sweeping nodes i such that $j \prec i$.

When $k \not\prec i$ then $c_{jk}^{(i)} = c_{ik}^{(i+1)}$ according to the second branch in (3). Furthermore, $H_{ik}^{(i+1)}$ can only be updated when sweeping the (unique) successor i_1 of i, namely

$$c_{jk}^{(i)} = c_{ik}^{(i+1)} = c_{i_1 k}^{(i_1)}, \text{ where } i_1 \succ i. \tag{4}$$

When $k \prec i$, the third branch in (3) indicates that $c_{jk}^{(i)}$ is at (most) $c_{ik}^{(i+1)} + c_{ij}^{(i+1)} + c_{ii}^{(i+1)} + 1$.

We claim that $c_{ik}^{(i+1)} = c_{ij}^{(i+1)} = 0$. To prove this, observe that $c_{ik}^{(i+1)} = c_{i_1 k}^{(i_1)}$ where $i_1 \succ i$; also, $c_{i_1 k}^{(i_1)}$ does not grow when sweeping i_1 because $k \not\prec i_1$ since k's only successor is $i \neq i_1$. Following the same argument we obtain that $c_{ik}^{(i+1)} = c_{i_1 k}^{(i_1)} = \ldots = c_{lk}^{(l)} = 0$. Similarily one can prove that $c_{ij}^{(i+1)} = 0$.

In conclusion, we have that that $c_{jk}^{(i)} \leq c_{ii}^{(i+1)} + 1$ when $k \prec i$. Since $c_{ii}^{(i+1)}$ can only be updated at the successor i_1 of i, we can also write

$$c_{jk}^{(i)} \leq c_{ii}^{(i_1)} + 1, \text{ where } i_1 \succ i. \tag{5}$$

The growth equations (4) and (5) indicate that when j and k are dependent nodes, the cardinality of (j,k) entry grows by at most one only when the nodes on the (unique) path from $i \succ j$ to l are swept. Therefore, $c_{jk}^{(1)}$ is at most the length of the path from i to l, which we denote by $\alpha(i)$. With this notation we can write

$$c_{jk}^{(1)} \leq \alpha(i), \text{ where } i \succ j. \tag{6}$$

On average $\alpha(i) = \log(l)$, which indicates a modest growth in the size of the (j,k) entry for dependent nodes j and k.

Case (ii): $j \geq 1$ **and** $k < 1$**.** There can be only one update in (3), at node $i \succ j$ since $j \geq 1$ has only one successor. If the update occurs in the second branch ($k \not\prec i$), then $c_{jk}^{(i)} = c_{ik}^{(i+1)} = c_{ik}^{(i_1)}$, with the last equality given by the fact that (i,k) can be only updated at $i_1 \succ i$.

We bound $c_{ik}^{(i_1)}$. First note that $c_{ik}^{(i_1)}$ is either $c_{i_1k}^{(i_1+1)}$ or $c_{i_1k}^{(i_1+1)} + c_{i_1i}^{(i_1+1)} + c_{i_1i_1}^{(i_1+1)} + 1$ accordingly to (3), for when $k \not\prec i_1$ and $k \prec i_1$, respectively. As in Case (i), one can prove that $c_{i_1i}^{(i_1+1)} = 0$. Also, $c_{i_1i_1}^{(i_1+1)} \leq \alpha_{i_1}$ by (6). As a result, the cases of both $k \not\prec i_1$ and $k \prec i_1$ have a common bound $c_{i_1k}^{(i_1)} \leq c_{i_1k}^{(i_1+1)} + \alpha(i_1) + 1 = c_{i_1k}^{(i_2)} + \alpha(i_1) + 1$, where $i_2 \succ i_1$. This last result shows that the bound can grow by at most by $1 + \alpha(p)$ at the nodes p of the (unique) path from i_1 to l for which $k \prec p$. Assuming the conservative situation that k is a predecessor of all such nodes, we obtain that

$$c_{jk}^{(1)} \leq s(k)(1 + \alpha). \tag{7}$$

The conservative situation corresponds to functions that have the independent variable k nested inside nonlinear operators, for example $f(v_{-n}, \ldots, v_0) = v_k \text{ op } f_1(v_k \text{ op } f_2(v_k, \ldots), \ldots)$.

If the update of $H_{jk}^{(i)}$ occurs in the third branch ($k \prec i$), then $c_{jk}^{(i)}$ is at most $c_{ik}^{(i+1)} + c_{ij}^{(i+1)} + c_{ii}^{(i+1)} + 1$. By the same reasoning used for (7), the first term can be bounded by $s(k)(1 + \alpha)$. As in Case (i), since $j \prec i$, the second term is zero; also, the third term is at most $\alpha(i)$ by (6). Therefore

$$c_{jk}^{(1)} \leq s(k)(1 + \alpha) + \alpha(i) + 1 \leq (s(k) + 1)(1 + \alpha). \tag{8}$$

Since bound in (8) is larger than the bound in (7), (8) is the worst-case bound for Case (ii).

Case (iii): $j < 1$ and $k \geq 1$. Note that j can have multiple successors and the (j, k) entry can be potentially updated every time when a successor of j is swept.

We consider first the second branch in (3), that is $k \not\prec i$. For this case we observe that $c_{jk}^{(i)} = c_{ik}^{(i+1)} \leq \alpha(i)$, with the last equality given by (6). The maximum growth is obtained by summing over all successors of j, namely $c_{jk}^{(1)} \leq \sum_{i:j\prec i} \alpha(i)$. Let $\alpha = \max\{\alpha(i) : i = 1, \ldots, l\}$. Then we have $c_{jk}^{(1)} \leq \alpha s(j)$.

For the third branch of (3), that is $k \prec i$, we observe that the update happens only once since k has only one successor. Consequently, $c_{jk}^{(i)} = c_{ik}^{(i+1)} + c_{ij}^{(i+1)} + c_{ii}^{(i+1)} + 1$. Note that $c_{ik}^{(i+1)} = 0$ since k is an dependent node and $k \prec i$ (see Case (i)) and $c_{ii}^{(i+1)} = \alpha(i_1)$ based on (6). Hence, $c_{jk}^{(i)} \leq c_{ij}^{(i+1)} + \alpha(i_1) + 1 = c_{ij}^{(i_1)} + \alpha(i_1) + 1$. Applying this argument recursively, one can obtain, similarly to (8), that

$$c_{jk}^{(1)} \leq (s(j) + 1)(1 + \alpha) \tag{9}$$

As in Case (ii), the bound is attained for functions that have the independent variable j nested inside nonlinear operators. The bound given by (9) for the case $k \prec i$ is larger than the bound obtained for the case $k \not\prec i$ and can be used as a worst-case bound for Case (iii).

Case (iv): $j, k < 1$. Let us first denote by $s(j, k)$ the number of common successors of j and k.

In (3), there can be at most $s(j) - s(j, k)$ updates on the second branch and at most $s(j, k)$ updates on the third branch. Consequently, based on the bounds obtained in Cases

(i), (ii), and (iii) we see that

$$\begin{aligned} c_{jk}^{(1)} &\leq (s(j) - s(j,k))s(k)(1+\alpha) + s(j,k)\left[s(k)(1+\alpha) + s(j)(1+\alpha) + \alpha + 1\right] \\ &= \left(s(j)s(k) + s(j,k)s(j)\right)(1+\alpha) + s(j,k)(1+\alpha) \end{aligned} \tag{10}$$

According to (6), (8), (9), and (10), the growth in the size of each of set-valued entries (j,k) of the Hessian is modest, being bounded by the product of a quadratic or linear term in the number of the successors of j and k and the "height" of the tree subgraph of the computational graph formed by the dependent nodes. The modest growth in the Hessian entries was initially counterintuitive to us, because at first glance Algorithm 1 seems to indicate that the size of the Hessian entry grows as a geometric series.

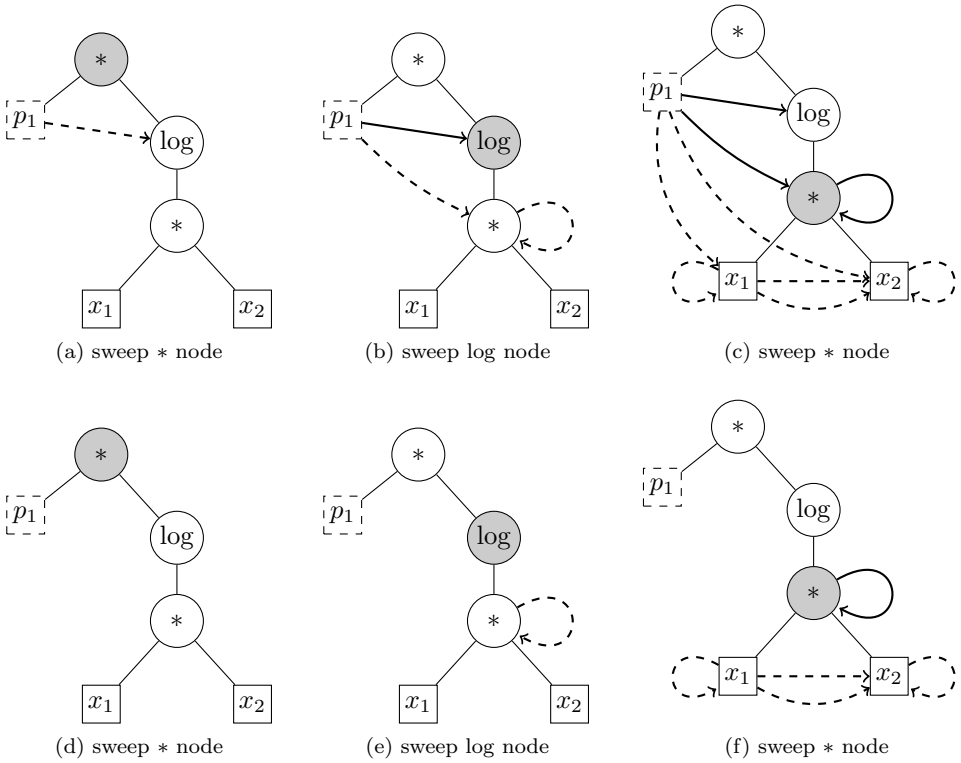
Note that this growth estimate is in terms of the computational graph, not in terms of the Hessian graph model; however, it is derived under the assumption that all the relationships between the nodes in the computational graph are nonlinear, which makes it very conservative.

We observe that the time complexity of computing entry (j,k) equals the space complexity, since the access and update times are $O(1)$. The approach used here to obtain complexity estimates is Hessian entry-centric and is different from the analysis in previous work ([1] and [3]), which is iteration-centric. In Gower and Mello [1], the time analyis is also expressed in terms of the (final) Hessian graph model; in Wang et al. [3] the analysis uses a measure of the nonlinear interactions between variables. For these reasons a direct comparison of our estimate with previous work is difficult. We remark that the complexity estimate of Wang et al. [3] is for the number of "create" and "push" steps and does not include a log term that comes from use of *std::map*. Also, the estimates of Gower and Mello [1] assume the use adjacency lists that require certain ordering on indexes of variables to obtain the stated complexity; such ordering does not necessarily occur in their ADOL-C implementation (reported to not work correctly by Wang et al. [3]).

3.3 *Implementation details*

We use a tape representation for the computational graph that is based on the Julia `Vector` type. This array-based tape implementation allows us to minimize the stack memory usage when the algorithm is implemented using loops. Initially, the edge pushing algorithm perform a sweep of the computational graph to count the number of elements in each of the Hessian entries and then allocate the space for the data structure. This preprocessing step is needed to avoid repeatedly growing the Julia vectors used to store the Hessian. The Hessian data structure can then be reused for subsequent Hessian evaluations during the optimization phase, therefore its cost, which is usually low, is amortized over the optimization iterations.

During the performance profiling phase in our implementation we identified a number of ways to speed up the Hessian evaluation and reduce storage requirements for real-world problems. The implementation of a simple strategy for pruning the parameters nodes in the Hessian graph model and an alteration of the tape structure for the specific case of AD applied to constrained optimization problems resulted in considerable improvement (as much as several factors) in the code performance. These two developments are presented next.



(a) sweep $*$ node (b) sweep log node (c) sweep $*$ node

(d) sweep $*$ node (e) sweep log node (f) sweep $*$ node

Figure 1.: Comparison of the edge pushing algorithm without ((a)-(c)) and with ((d)-(f)) pruning the parameter node p_1 for function $p_1 \cdot \log(x_1 \cdot x_2)$. Circles represent dependent nodes, dotted squares represent parameter nodes, and solid squares represent independent nodes. The dotted edges indicate newly created edges when sweeping a incumbent node, which is marked with gray color. Pruning reduces the number of edges from 9 to 5.

3.3.1 Pruning the Hessian graph model

The general strategy in the edge pushing algorithm consists of visiting the computational graph starting at the node corresponding to the final dependent variable v_l and updating the Hessian graph model according to recurrence (3) by *creating* and *pushing* new edges. These edges correspond to nonlinear relationships between the variables, which can potentially contribute to the entries in Hessian. The "creating" step discovers new nonlinear relationships and updates the current node's weight accordingly; the "pushing" step propagates such nonlinear relationships to each of the predecessors of the the current node and creates shortcuts edges.

For the purpose of computing zeroth- (function evaluation) and first-order (gradient) derivatives, the parameters in the original function expression are treated as dependent variables and appear in the computational graph (and thus in Hessian graph as well) as regular nodes. A critical observation is that in the Hessian computation, parameters do not influence the nonlinear relationships between the other variables. In terms of the Hessian graph model, the edges that have at least one parameter node as endpoint do not need to be "pushed" since they will have no second-order derivative contribution to the final Hessian value. Similarly, in the "creating" step, the edges ending in a parameter node are discarded. We illustrate this technique in Figure 1. This strategy of pruning the Hessian graph can considerably reduce the number of edges in the Hessian graph

model. The reduction in space requirements can be drastic, as much as several times, for problems that have many parameters nodes close to the top dependent node of the computational graph. Time savings are similar, stemming from a reduced number of edges that needs to be pushed or created.

3.3.2 Optimization-specific developments

When used in an algebraic modeling language for optimization, the edge pushing implementation needs to provide the function and gradient evaluations individually for the objective f and each constraint g_i, $i = 1, \ldots, m$, which requires the tape to be built for each of these functions. On the other hand, the algebraic modeling language does not require the individual Hessians of f and g_i; instead it require the Hessian of the Lagrangian, which has the form

$$L(x) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x),$$

where the parameters λ_i are specific to the optimization algorithm.

In our implementation we build the tape directly for the computational graph of the Lagrangian and keep track where the computational graph of f and each g_i starts and ends (in two arrays of size m) in the tape. Note that due to additive form of L, the tape representations of the computational of graphs of f and g_i are sequential on the tape corresponding to L. This strategy makes it possible to evaluate f and g_i, as well as their gradients, individually as required by the algebraic modeling language; it also allows us to compute the Hessian of the Lagrangian directly, without computing the intermediary Hessians of the objective and constraints.

3.4 *Final remarks*

It is well known that Edge Pushing algorithms are designed to implicitly take advantage of sparsity [1, 3]. To this extent, we remark that our EP variant does not alter this salient property of EP algorithms; indeed, (3) indicates that no additional nonzero entries in the Hessian occurs by allowing set-valued entries since these occur only for nonzero partials derivatives, which are the same in our algorithm as in its sibling [1]. In other words, the Hessians computed by the two algorithms are identical in terms of sparsity pattern (in addition to having identical numerical values).

Partially separable functions [9, 10] arise in many optimization problems and proved to improve the performance of numerical optimization algorithms [11, 12]. It is important to remark that partial separability is exploited *implicitly* by the class of Edge Pushing algorithms as the byproduct of their implicit sparsity detection combined with the fact that that separable operators + and − are linear and, therefore, they do not introduce nonlinear relationships among their operands during the pushing and creating steps. This results in a separable computational graph. We illustrate this in Figure 2 for the separable function of the form

$$\sum_{i=1}^{M} \log\left(1 + \frac{1}{\exp(y_i \sum_{j=1}^{N} \theta_j x_{ij})}\right). \tag{11}$$

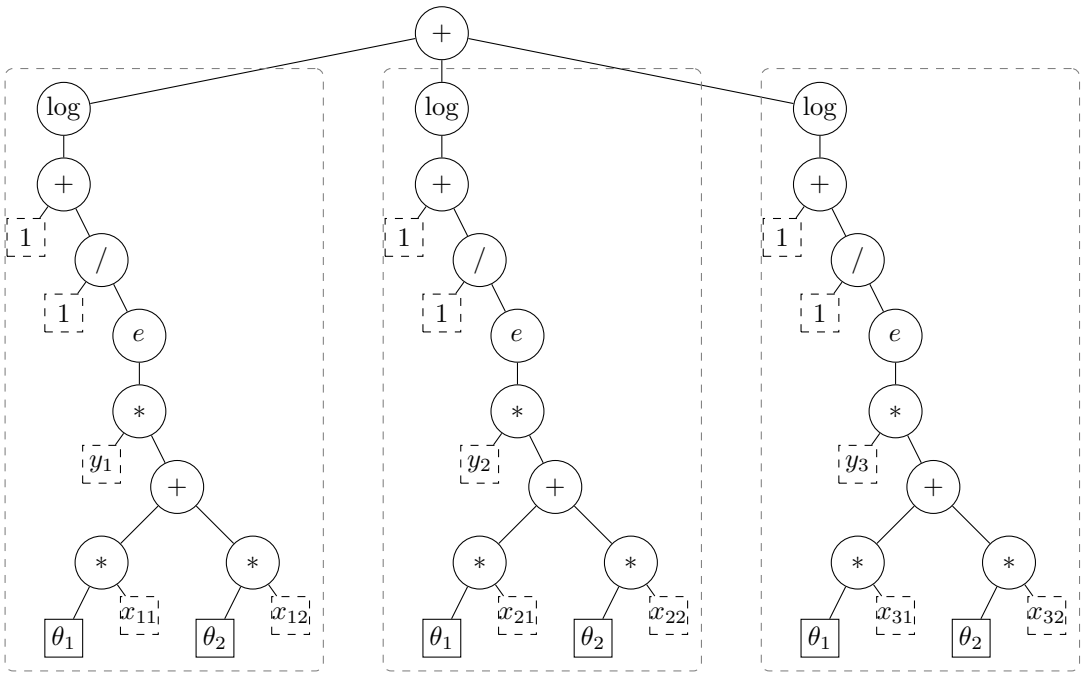


Figure 2.: The computational graph of the separable function given by (11) for $N = 2, M = 3$. EP algorithms implicitly take advantage of separability by sweeping only the subgraphs rooting in the separable operator +. Partially separable functions are similar. Parameter nodes are represented by dashed boxes, independent nodes by solid boxes, and dependent nodes by circles.

4. Computational experiments and performance results

We compare the performance of the edge pushing implementation with the Julia coloring-based Hessian computation implementation from JuMP, and also with the AD of the commercial algebraic modeling language AMPL. We report the average Hessian evaluation time, denoted by "Hess.", and the total auto differentiation time, denoted by "AD", over 5 optimization iterations. All the times are reported by Ipopt 3.12.5. We also record the replication factor (rf) of the Hessian entries within our edge pushing algorithm, which is computed as the ratio of the sum of cardinals of the set-valued Hessian entries (corresponding to independent nodes) and the number of nonzero elements in the Hessian. This replication factor effectively characterizes the amount of extra memory required by our algorithm. To ensure a fair comparison of the algorithms, we do not report coloring times of JuMP for problems for which the coloring is trivial *a priori*, e.g. for arrow-head and dense Hessians [2, 6–8, 13], even though JuMP currently uses general-purpose coloring methods and does not take advantage of these special structures. The preprocessing step of our EP implementation, that is, the sweep that prealocates the Hessian data structure, is an small fraction of the Hessian evaluation time and is not recorded. On the other hand, the postprocessing step, which sum up the duplicate entries of each Hessian entry in the buffer offered by the optimization solver is recorder in the Hessian evaluation time (Hess.).

The numerical experiments and performance evaluation were performed on RedHat

Linux machine running a Intel© Xeon© CPU E5-1650 v3 at 3.50GHz with 64Gb of RAM memory. We used Julia v0.4, JuMP v0.14.2, and AMPL v20161231. Our implementation was checked for correctness by comparing it with JuMP on several instances of each the class of test problems; Ipopt reported identical solution and iteration convergence metrics (within machine precision and optimization convergence criteria).

The problem classes used in the performance evaluation were chosen for the dual purpose of (i) scrutinizing the performance of the Julia vector-based implementation against our theoretical expectations of fast access time and modest replication factor and (ii) comparing the performance of our EP implementation with coloring-based AD of JuMP and state-of-the-art algebraic modeling language AMPL to assess any potential improvement for Julia AD users. In particular, the synthetic examples of Section 4.1 and 4.2 serve primarily as testbed for the study of the dependence of the execution time and storage on the Hessian size and on the number of nonlinear relationships among the variables. The log-regression class of problem 4.3 are used to stress-test the proposed Hessian data structure for dense Hessians. Finally, the power grid problems represent a realistic testbed to evaluate the performance gap between current Julia AD capabilities in algebraic modeling languages and state-of-the-art algebraic modeling AMPL.

4.1 *Arrow-head structure*

The first set of experiments are performed for functions of the form

$$\sum_{i=1}^{N}\left[\cos\left(\sum_{j=1}^{K} x_{i+j}\right)+\sum_{j=1}^{K}(x_i+x_j)^2\right], \tag{12}$$

which have Hessians (of size N) with K dense bordering columns/rows, *i.e.*, arrow-head shaped Hessians. We perform two sets of experiments. The first, shown in Table 1, measures the performance of the three approaches for Hessians of increasing sizes with a fixed number of bordering columns and rows. The speed-up with N is linear for all three approaches; edge pushing and AMPL performance is similar for both Hessian evaluation and AD time.

For the second set of experiments, shown in Table 2, the size of the Hessian is fixed and the bordering size is increased. The computation time of all three approaches grows faster than linear when K is increased. In the case of edge pushing, this behavior is caused by the increasingly large number of nonlinear relationships between variables in (12) when K grows; as a result, the replication factor also increases with K. We note that the replication factor grows slower than K. For this set of experiments, edge pushing seems to match the performance of AMPL. On the other hand, JuMP's performance for these sets of problems is affected by the quality of coloring. We remark that coloring for arrow-head Hessians can be improved by first computing the coloring of the submatrix obtained by removing the border, and adding the border dense columns to this coloring. This observation is very likely to improve JuMP's performance.

4.2 *Random sparsity structure*

The second performance evaluation is performed on problems with sparse *unstructured* Hessians. These problems were synthetically generated to have an arbitrary sparsity

Table 1.: The performance results for arrow-head Hessians of increasing size with fixed number of bordering columns and rows ($K = 16$).

N	K	nnz	Edge pushing rf	Edge pushing Hess. time	Edge pushing AD time	JuMP Hess. time	JuMP AD time	AMPL Hess. time	AMPL AD time
2000	16	63760	5.7	0.005	0.079	0.036	0.040	0.003	0.007
4000	16	127760	5.7	0.009	0.086	0.074	0.079	0.006	0.014
8000	16	255760	5.7	0.019	0.102	0.151	0.157	0.016	0.031
16000	16	511760	5.7	0.040	0.133	0.300	0.310	0.038	0.067
32000	16	1023760	5.7	0.080	0.194	0.603	0.619	0.070	0.124
64000	16	2047760	5.7	0.158	0.318	1.229	1.229	0.157	0.263

Table 2.: The performance results for arrow-head Hessians of increasing bordering size with function given in (12), when $N = 32\,000$ is fixed.

N	K	nnz	Edge pushing rf	Edge pushing Hess. time	Edge pushing AD time	JuMP Hess. time	JuMP AD time	AMPL Hess. time	AMPL AD time
32000	2	127998	2.2	0.010	0.090	0.014	0.019	0.006	0.018
32000	4	255988	2.7	0.017	0.111	0.039	0.046	0.010	0.027
32000	8	511944	3.7	0.035	0.130	0.161	0.172	0.026	0.056
32000	16	1023760	5.7	0.079	0.195	0.622	0.639	0.069	0.124
32000	32	2047008	9.7	0.236	0.389	2.091	2.124	0.177	0.280
32000	64	4091968	17.7	0.667	0.907	7.569	7.623	0.411	0.623

pattern by considering objective functions of the form

$$\sum_{i=1}^{N} \left[(x_i - 1)^2 + \prod_{j \in \text{rand\_set}_i(N,K)} x_j \right]. \tag{13}$$

The dimension of the Hessian matrix is N, while K controls the density of the Hessian. The random sparsity pattern is obtained by randomly generating a set $\text{rand\_set}_i(N, K)$ of K unique indexes from $\{1, 2, \ldots, N\}$ over which the the multiplication operator is applied. This set is generated for each term $i \in \{1, 2, \ldots, N\}$. This strategy places nonzero elements in each row and column of the Hessian at random positions. Additional nonzero entries occur on the diagonal of the Hessian of (13) from the quadratic terms $(x_i - 1)^2$.

We are interested in the performance of the three AD approaches for this set of problems because the unstructured sparsity of the Hessians require the use of coloring in JuMP. In addition, AMPL is very likely to employ its algorithm for general sparse Hessian computation [14].

Tables 3 and 4 show the dimensions of the benchmark problems and the performance results for fixed N and fixed K, respectively. For any given values of N and K, the same random index sets $\text{rand\_set}_i(N, K)$ were used in comparing edge pushing, JuMP's AD, and AMPL. We first note from Table 4 that all three approaches show linear time dependence with N for fixed K. On the other hand, when K varies and N is fixed, the number

Table 3.: The performance results for (13) for fixed $N = 4000$.

N	K	nnz	Edge pushing rf	Edge pushing Hess. time	Edge pushing AD time	JuMP Hess. time	JuMP AD time	JuMP coloring time	AMPL Hess. time	AMPL AD time
4000	2	7999	1.0	0.001	0.073	0.001	0.004	0.024	0.001	0.002
4000	4	27936	1.0	0.001	0.073	0.005	0.008	0.038	0.002	0.003
4000	8	115104	1.0	0.003	0.075	0.023	0.027	0.155	0.008	0.010
4000	16	468041	1.0	0.011	0.083	0.264	0.269	1.821	0.031	0.035
4000	32	1749489	1.1	0.078	0.156	3.395	3.401	26.646	0.146	0.156
4000	64	5041468	1.6	0.572	0.670	16.576	16.587	188.558	0.591	0.608

Table 4.: The performance results for function given in (13), when fixing $K = 32$.

N	K	nnz	Edge pushing rf	Edge pushing Hess. time	Edge pushing AD time	JuMP Hess. time	JuMP AD time	JuMP coloring time	AMPL Hess. time	AMPL AD time
1000	32	309788	1.6	0.018	0.089	0.311	0.3156	2.440	0.027	0.029
2000	32	774325	1.2	0.037	0.112	1.154	1.158	9.838	0.049	0.053
4000	32	1749489	1.1	0.076	0.154	3.401	3.407	26.844	0.143	0.153
8000	32	3727099	1.0	0.162	0.247	7.126	7.135	65.159	0.371	0.392
16000	32	7697347	1.0	0.313	0.416	12.883	12.900	136.839	0.779	0.817

of nonzeros grows quadratically as shown in Table 3; edge pushing and AMPL running times (both Hessian computation and total time) seem to grow quadratically with K as shown in Table 3. We remark that the number of nonzeros also grows quadratically. JuMP's Hessian times seem to grow at a faster rate, which can be explained by a possible degradation of the coloring quality as K grows (due to denser Hessians).

Edge pushing seems to outperform AMPL by as much as a factor of 2 (Table 4). This performance gap seems to close and even disappear for problems where edge pushing's replication factor increases, as it can be seen in the last rows of Table 3 JuMP's performance appears to be significantly affected by the unstructured sparsity, for which the coloring proves to be both expensive, and as mentioned in the previous paragraph, of low quality.

4.3 *Logistic regression models*

Logistic regression is used in statistics and computer science (e.g., machine learning, classification, natural language processing) to conduct regression analysis when the dependent variable is binary. The training of the parameters (denoted by an N-dimensional vector θ) in logistic regression is done by solving

$$\min_{\theta} \lambda\|\theta\|^2 + \sum_{i=1}^{M} \log\left(1 + \exp\left(-y_i \theta^T x_i\right)\right), \tag{14}$$

where $\{x_i, y_i\}_{i=1}^{M}$ ($x_i \in \mathbb{R}^N$, $y_i \in \mathbb{R}$) are the training points and λ is a constant chosen to be inversely proportional with the covariance of the prior distribution of θ [15].

Table 5.: Performance results for logistic regression models.

M	N	Edge pushing rf	Edge pushing Hess. time	Edge pushing AD time	JuMP Hess. time	JuMP AD time	AMPL Hess. time	AMPL AD time
2	2000	6.0	0.033	0.105	0.067	0.070	0.004	0.004
2	4000	6.0	0.224	0.295	0.277	0.280	0.015	0.015
2	6000	6.0	1.119	1.192	1.270	1.273	0.052	0.052
2	8000	6.0	5.024	5.096	5.957	5.960	0.211	0.211

A quick inspection of the model reveals that the Hessian (of size $N \times N$) is dense, thus logistic regression models would serve as great "stress test" instances that can potentially reveal potential limitations or performance issues in the edge pushing algorithm and our implementation. For this reason we have taken N larger than it might be in typical applications.

Table 5 shows the performance evaluations for (14) for increasing values of Hessian dimension N. The Hessian evaluation times of edge pushing, JuMP, and AMPL seem to increase quadratically with the N, which is expected since the Hessian is dense (thus the number of nonzeros grows quadratically).

Edge pushing is slightly faster than the coloring algorithm; however, the space requirement is roughly six times larger because of our replication strategy. On the other hand, AMPL is considerably faster than both edge pushing and coloring. It is unclear to us what algorithm AMPL uses for dense Hessian computations. The work published in [14] seems to indicate that summation of (dense) outer products are at the core of AMPL's Hessian computation. We believe that AMPL's reliance on dense linear algebra kernels and the use of sparse data structures in edge pushing and coloring implementations, which incur a considerable performance penalty for dense matrices because of the irregular memory access and time overhead from the integer arithmetic, could explain the performance gap. For example, during the performance profiling phase of our implementation, we observed that the running time of summing and copying the duplicated Hessian entries in the buffer offered by the optimization solver at the end of the edge pushing algorithm takes about 60% of the **total** AMPL Hessian evaluation time ($N = 8\,000$). To rule out potential Julia overhead we replicated the same summation/copying loop in C++ (with *-O3* optimization); roughly the same running time was obtained. In the light of this experiment, it is apparent to us that efficient computations of dense Hessians using the edge pushing algorithm requires at minimum a reconsideration and redesign of the Hessian data structure.

4.4 *Power grid models*

The last set of experiments are performed on alternating current optimal power flow (ACOPF) problems. Such models are used on a daily basis in power grid operations as optimization-based procedures for dispatching electricity in both the transmission and distribution systems; they also serve as the backbone for the calculation of short-term and realtime electricity prices. We use the ACOPF model made available by Artelys as

a benchmark problem in AMPL format. The model was translated in JuMP format in previous work by authors [4] and published as a benchmarking problem [16] for JuMP. The mathematical formulation of the Kirchoff's laws of power flow uses polar coordinates, resulting in a highly nonlinear model. In addition, the networked nature of the problem produces a Hessian that is highly structured, making this realistic problem a challenging test for Hessian AD algorithms.

Table 6.: Performance results for nonlinear AC optimal power flow problems.

			Edge pushing			JuMP			AMPL	
Buses	Edges	nnz	rf	Hess. time	AD time	Hess. time	AD time	coloring time	Hess. time	AD time
662	1017	8121	13.8	0.004	0.079	0.008	0.014	0.045	0.002	0.005
6620	10170	812100	13.8	0.046	0.135	0.086	0.132	0.718	0.032	0.066
66200	101700	8121000	13.8	0.457	0.702	0.882	1.324	16.584	0.319	0.6474

In Table 6 we show the performance of edge pushing, JuMP, and AMPL on three ACOPF instances. The first instance uses a network with 662 nodes and 1 017 edges and has 1 489 decision variables and 1 324 constraints. The Hessian of the Lagrangian has 8 121 nonzero entries. The other two instances were obtained synthetically by replicating the network by 10 and 100 times, which results in a proportional increase in the the dimension of the optimization problem. These two instances are also available at [16].

As expected, total AD times seem to grow linearly with the number of nonzeros in the Hessian. Edge pushing and AMPL perform the best and are quite close, while coloring-based algorithm of JuMP is roughly within a factor of two. Note that the coloring time within JuMP is significant for this family of instances with an unstructured sparsity pattern.

5. Conclusions

We proposed a data structure for efficient computations of large-scale sparse Hessians in Julia using the Edge Pushing algorithm. The Julia implementation of the proposed variant of the Edge Pushing algorithm is successful in substantially closing the gap between JuMP's existing implementation of Hessian computations and that of AMPL. These results call for making the EP algorithm easily available for users, perhaps completely replacing JuMP's current coloring-based approaches. A considerable gap still exists for dense Hessians, for which we advocate for the use of a specialized, "dense" data structure with the Edge Pushing algorithm and possibly using a different EP basecode, following AMPL's approach.

Supplementary materials

The source code of our EP implementation is available at `https://github.com/fqiang/ReverseDiffTape.jl`. The models that are used in our benchmark results in Section 4 are available in the benchmark directory of this repository.

Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and by Argonne National Laboratory under Contract No. DE-AC02-06CH11357. We thank Alex Pothen, Mihai Anitescu, Jarrett Revels, and the anonymous referees of a previous version of this manuscript for valuable suggestions and guidance.

References

[1] R. M. Gower and M. P. Mello. A new framework for the computation of Hessians. *Optimization Methods and Software*, 27(2):251–273, 2012.

[2] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther. Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21(2):209–223, 2009.

[3] M. Wang, A. Gebremedhin, and A. Pothen. Capitalizing on live variables: new algorithms for efficient Hessian computation via automatic differentiation. *Mathematical Programming Computation*, pages 1–41, 2016.

[4] I. Dunning, J. Huchette, and M. Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 2017 (to appear).

[5] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial Mathematics, 2nd edition, November 2008.

[6] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

[7] Assefaw H Gebremedhin, Arijit Tarafdar, Fredrik Manne, and Alex Pothen. New acyclic and star coloring algorithms with application to computing hessians. *SIAM Journal on Scientific Computing*, 29(3):1042–1072, 2007.

[8] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, October 2013.

[9] A. Griewank and Ph. L. Toint. Local convergence analysis for partitioned quasi-Newton updates. *Numerische Mathematik*, 39(3):429–448, 1982.

[10] A. Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39(1):119–137, 1982.

[11] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[12] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust-region Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[13] Alex Pothen. personal communication, 2016.

[14] David M Gay. More AD of nonlinear AMPL models: computing Hessian information and exploiting partial separability. *Computational Differentiation: Applications, Techniques, and Tools*, pages 173–184, 1996.

[15] A Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in neural information processing systems*, 14:841, 2002.

[16] I. Dunning, J. Huchette, and M. Lubin. JuMPSupplement. `https://github.com/mlubin/JuMPSupplement`, 2016.