An Efficient Automatic Differentiation Algorithm for Hessians: Working With *Live* Variables[*]

Mu Wang Assefaw Gebremedhin Alex Pothen

Introduction. Gower and Mello [3] recently introduced a graph model for computing Hessians using Automatic Differentiation (AD) [1, 2]. In the model, which is based solely on Reverse Mode AD, the computational graph of the input function is augmented with minimal information—additional edges corresponding precisely with nonlinear interactions—and the symmetry available in Hessian computation is exploited. Using the model, they developed an algorithm, called `Edge_Pushing`, where edges representing nonlinear interactions are 'created' and their contributions to descendants are 'pushed' as the algorithm proceeds. The approach represents an important progress in AD for Hessians, but it unfortunately has several shortcomings. First, the authors' derivation of the algorithm is rather complicated and hard to understand. Second, their implementation in `ADOL-C` relies on an indexing assumption not necessarily supported by `ADOL-C` so that their code gives incorrect results in some cases. In this work, we provide a new, intuitive derivation of the `Edge_Pushing` algorithm from a completely different perspective and a robust implementation (built on top of `ADOL-C`) that works correctly in all cases. At the heart of our approach lies this: we identify an invariant in the first order *incremental reverse mode* of AD, which we arrive at by taking a *data-flow* perspective. We obtain the Hessian algorithm by extending the invariant to second order. Additionally, we incorporate *preaccumulation* in the Hessian algorithm to further enhance performance.

Reverse Mode AD. In data-flow analysis in compiler theory, a variable is said to be *live* if it holds a value that *might be* read in the future. We find a similar notion useful in our context. Since in reverse mode AD all information about the execution sequence of the objective function is recorded on an evaluation trace, we can in fact work with a more restricted definition for a live variable. In particular, we say a variable is live if it holds a value that *will be* read in the future. And we call the set made up of all live variables at each step of the execution sequence a *live variable set* in that sequence.

Following the notations of Griewank and Walther [1], the first order incremental reverse mode of AD can be written using a sequence of Single Assignment Code (SAC) as:

```
Algorithm: First Order Incremental Reverse Mode (FOIRM)
Initialization: $\bar{v}_l = 1.0$, $\bar{v}_{1-n} = \cdots = \bar{v}_0 = \bar{v}_1 = \cdots = \bar{v}_{l-1} = 0$
for $i = l, \cdots, 1$ do
    for all $v_j \prec v_i$ do                ($\prec$ denotes precedence)
        $\bar{v}_j += \frac{\partial \varphi_i}{\partial v_j} \bar{v}_i$
```

We observe the following invariant in this mode of computing adjoints:

Observation 1 *The set of adjoints computed in each step i of the FOIRM algorithm involve partial derivatives with respect to only the current live variable set, not the entire set of variables.*

Hessian Algorithm. We extend the invariant formulated in Observation 1 to second order derivatives. Let S denote the set of live variables in a given step. Then, the first order derivatives (adjoints), denoted in the code in FOIRM by $\bar{v}$, for each $v \in S$, can be viewed as a mapping $a : S \to \mathbb{R}$, $a(v) = \bar{v}$. Analogously, the second order derivatives (Hessian) can be viewed as a symmetric mapping $h : S \times S \to \mathbb{R}$, $h(v, u) = h(u, v)$. Our target algorithm is then precisely a prescription of how S, $a(S)$ and $h(S, S)$ should be changed as the associated SAC is processed such that the invariant is maintained.

Let $\hat{S}$, $\hat{a}(\hat{S})$ and $\hat{h}(\hat{S}, \hat{S})$ denote the live variable set, the adjoint mapping, and the Hessian mapping, respectively, after a SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ is processed. Because the sequence proceeds in the reverse order,

[*]A full report on this work is being submitted elsewhere. Authors' emails: {wang970, agebreme, apothen}@purdue.edu

we have $\hat{S} = \{S \setminus \{v_i\}\} \cup \{v_j | v_j \prec v_i\}$. Considering the adjoints equation, and noting that $\frac{\partial \varphi_i}{\partial v_j} = 0$ when $v_j \not\prec v_i$, and $a(v_j) = 0$ when $v_j \notin S$:

$$\forall v_j \in \hat{S}, \hat{a}(v_j) = a(v_j) + \frac{\partial \varphi_i}{\partial v_j} a(v_i).$$

For the second order rule, noting that $h(v_j, v_k) = 0$ when $v_j \notin S$ or $v_k \notin S$, and applying the chain rule of calculus, analogous to the adjoint case, we have $\forall v_j, v_k \in \hat{S}$:

$$\begin{aligned}\hat{h}(v_j, v_k) = h(v_j, v_k) + \frac{\partial \varphi_i}{\partial v_j} h(v_i, v_k) + \frac{\partial \varphi_i}{\partial v_k} h(v_i, v_j) + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial \varphi_i}{\partial v_k} h(v_i, v_i) \\ + a(v_i)\frac{\partial^2 \varphi_i}{\partial v_j \partial v_k}. \qquad (1)\end{aligned}$$

Equation (1) corresponds to the `Edge_Pushing` algorithm of [3], in which the last three terms on the first line represent the pushing part, and the sole term in the second line represents the creating part in the component-wise form of their algorithm.

Implementation and Evaluation. We implemented this data flow-based Hessian algorithm in `ADOL-C`. We observe that in order to take advantage of the symmetry available in Hessian computation, the result variables in the SAC sequence need to have monotonic indices. However, the location scheme for variables currently used in `ADOL-C` does not satisfy this property, which is one reason why the Gower-Mello implementation of the `Edge_Pushing` algorithm fails. We implemented a fix in `ADOL-C` where we appropriately translate indices of variables before starting the reverse Hessian algorithm.

To further improve efficiency, we incorporate a statement-level *preaccumulation* technique to the Hessian algorithm. Preaccumulation splits the reverse Hessian algorithm into a local and a global level. In the local level, each SAC is processed to compute the first and second order derivatives of local functions defined by assign-statements in the execution path. In the global level, the derivatives of each local function is accumulated to compute the entire Hessian of the objective function.

The table below shows sample results comparing the runtime (sec.) of the new approach (`EPwithPreacc` and `EPwithoutPreacc`) with two related approaches: (i) a full Hessian algorithm in which sparsity is *not* exploited (`Full-Hessian`) and (ii) two compression-based sparse Hessian algorithms involving sparsity structure detection, graph coloring, compressed evaluation and recovery (`SparseHess-direct` and `SparseHess-indirect`). Results are shown for synthetic test functions from [5] and mesh optimization problems in the FeasNewt benchmark [4]. Details will be discussed in the upcoming full report.

	Synthetic				**Mesh Optimization**		
Matrix order n:	10,000	10,000	10,000	10,000	2,598	11,597	39,579
Number of nonzeros:	19,999	59,985	44,997	59,985	46,488	253,029	828,129
`Full-Hessian`	31.78	573.16	28.91	33.83	129.36	> 2 hours	> 2 hours
`SparseHess-direct`†	0.04	0.30	0.12	16.05	5.17	37.35	129.35
`SparseHess-indirect`†	0.20	0.31	0.33	25.72	4.14	28.94	111.97
`EPwithoutPreacc`	0.05	0.27	0.12	0.12	0.53	3.63	12.80
`EPwithPreacc`	0.06	0.23	0.08	0.10	0.48	3.27	11.10

† The times are a total of the four steps, whose contributions vary greatly. As an example, the breakdown for the largest mesh optimization problem (nnz=828,129) is:

	Pattern	**Coloring**	**Compressed H.**	**Recovery**
`SparseHess-direct`	54.1%	1.02%	44.8%	0.03%
`SparseHess-indirec`	61.1%	0.96%	24.8%	13.1%

References

[1] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* 2nd Edition, SIAM, 2008.

[2] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* SIAM, 2012.

[3] R. M. Gower and M. P. Mello. *A New Framework for The Computation of Hessians.* Optimization Methods and Software, Volume 27, Issue 2, pp 251–273, 2012.

[4] T. S. Munson and P. D. Hovland. *The FeasNewt Benchmark.* IEEE International Symposium on Workload Characterization (IISWC), 2005.

[5] L. Luksan, C. Matonoha and J. Vlcek: *Sparse Test Problems for Unconstrained Optimization.* Tech. Rep. V-1064, ICS AS CR, January 2010.