Storing versus recomputation on multiple DAGs

Heather Cole-Mullen, Andrew Lyons, and Jean Utke

Abstract Recomputation and storing are typically seen as a tradeoff for checkpointing schemes in the context of adjoint computations. At finer granularity during the adjoint sweep, in practice, only the store-all or recompute-all approaches are fully automated. This paper considers a heuristic approach for exploiting finer granular recomputations to reduce the storage requirements and thereby improve the overall adjoint efficiency without the need for manual intervention.

Key words: source transformation, reverse mode, storage recomputation tradeoff, heuristics

1 Introduction

Computing derivatives of a numerical model $\boldsymbol{f}:\boldsymbol{x}\mapsto\boldsymbol{y}:\mathbb{R}^n\mapsto\mathbb{R}^m$, given as a computer program P, is an important but also computation-intensive task. Automatic differentiation (AD) [6] in *adjoint* (or *reverse*) mode provides the means to obtain gradients and is used in many science and engineering contexts (refer to the recent conference proceedings [2, 1]). Two major groups of AD tool implementations are operator overloading tools and source transformation tools. The latter are the focus of this paper. As a simplified rule, for each intrinsic floating-point operation ϕ (e.g., addition, multiplication, sine, cosine) that is executed during runtime in P as the sequence

$$[\ldots, j:(\ u=\phi(v_1,\ldots,v_k)),\ldots],\quad j=1,\ldots,p, \tag{1}$$

Heather Cole-Mullen, Jean Utke
Argonne National Laboratory / The University of Chicago, IL, USA, `{hlcm|utke}@mcs.anl.gov`

Andrew Lyons
Dartmouth College, Hanover, NH, USA `lyonsam@gmail.com`

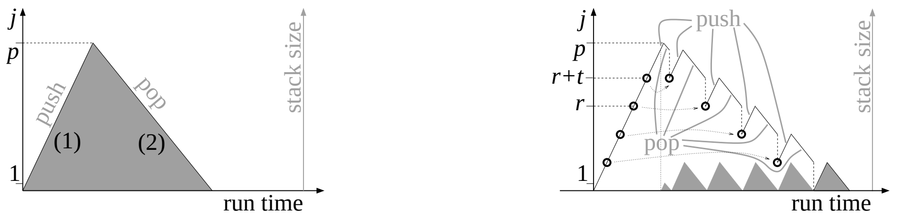


Fig. 1 Tape space for phases (1) and (2) without (left) and with (right) checkpointing.

of p such operations, the generated adjoint code has to implement the following sequence that reverses the original sequence in j:

$$[\ldots, j : (\,\bar{v}_1 {+}{=} \frac{\partial\phi}{\partial v_1}\bar{u}, \ldots, \bar{v}_k {+}{=} \frac{\partial\phi}{\partial v_k}\bar{u}), \ldots], \quad j = p, \ldots, 1, \tag{2}$$

with incremental assignments of adjoint variables $\bar{v}$ for each argument v of the original operation ϕ. If $m = 1$ and we set $\bar{\boldsymbol{y}} = 1$, then the adjoint sequence yields $\bar{\boldsymbol{x}} = \nabla \boldsymbol{f}$. The two phases are illustrated in Fig. 1; note that to compute $\frac{\partial\phi}{\partial v_i}$ in phase (2), one needs the values of the variables v_i from phase (1).

The need to store and restore variable values for the adjoint sweep requires memory, commonly referred to as *tape*, for the derivative computation. This tape storage can be traded for recomputations in a checkpointing scheme. In theory, the storage for the tape and the checkpoints may be acquired from one common pool, as was considered in [7]. However, practical differences arise from the typical in-memory stack implementation of the tape in contrast to the possible bulk write and read to disk for checkpoints. Furthermore, one may nest checkpoints or do hierarchical checkpointing [5]) while the tape access is in general stack-like. The impact of the taping on the checkpointing scheme and the overall adjoint efficiency is the size of the checkpointed segment of the program execution, which is limited by the available memory. *Reducing the storage requirements for taping implies a larger checkpointed segment, which implies fewer checkpoints written and read, which implies fewer recomputations in the hierarchical checkpointing scheme*.

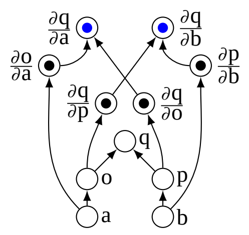
The goal of source code analysis has been the reduction of taping storage [8] for the “store-all” approach and the reduction of recomputation [3] for the “recompute-all” approach. The recompute-all approach replaces the tape altogether, at least initially, whereas the adjoint sweep requires the values to be available in the reverse order of the original computation. Recomputing the values in reverse order can carry a substantial cost. Consider a loop with k iterations and loop carried dependencies for the values to be recomputed. Computing the loop itself has a cost of k times the cost of a single iteration. Recomputing the values in reverse order has a complexity of $\mathcal{O}(k^2)$ times the cost a single iteration. In tool implementations, [4], this problem is mitigated by allowing the user to manually force certain, expensive-to-recompute values to be stored on a tape. This manual intervention can achieve an excellent tradeoff between taping and recomputation but requires deep insight into the code and is fragile in models that are subject to frequent changes.

Static source code analysis often cannot reliably estimate the complexity of recomputing values as soon as the computation includes control flow or subroutine calls. On the other hand, one can safely assume that re-executing a fixed, moderate-length sequence of built-in operations and intrinsic calls to recompute a given value will be preferable to storing and restoring said value. Such fixed, moderate-length sequences are given naturally by the computational graphs already used for the elimination heuristics in OpenAD [10].

Following the approach in [10], we denote the computational graph representing a section of straight-line code (think sequence of assignments) with $G^i = (V^i, E^i)$. The G^i are directed acyclic graphs with vertices $v_j \in V^i = V^i_{min} \cup V^i_{inter} \cup V^i_{max}$where $V^i_{min}$are the minimal vertices, $V^i_{max}$the maximal vertices and $V^I_{inter}$the intermediate vertices of G^i. The direct predecessors $\{\ldots, v_i, \ldots\}$ of each intermediate or maximal vertex v_j represent the arguments to a built-in operation or intrinsic $\phi(\ldots, v_i, \ldots) = v_j$. In the usual fashion, we consider the partials $\frac{\partial \phi}{\partial v_i}$ as labels $c_j i$ to edge (v_i, v_j).

Generally, these partials have a closed-form expression in terms of the predecessors v_i, and we can easily add them to the G^i. More flexible than the rigid order suggested by (2) is the use of elimination techniques (vertex, edge or face elimination) on the computational graph to preaccumulate the partial derivatives. The elimination steps performed on the G^i reference the edge labels as arguments to fused multiply-add operations, which themselves can be represented as simple expressions whose minimal vertices are edge labels or maximal vertices of (preceding) multiply-add operations. They too can be easily added to the G^i, and we denote the computational graph with the partial expressions and the preaccumulation operations as the extended computational graph G^{*i}. For the propagation of the adjoint

```
o=sin(a);
p=cos(b);
q=o*p;
```



$$\left.\begin{aligned} \frac{\partial o}{\partial a} &= \cos(a); \\ \frac{\partial p}{\partial b} &= -\sin(b); \\ \frac{\partial q}{\partial p} &= o; \\ \frac{\partial q}{\partial o} &= p; \end{aligned}\right\} \text{linearization}$$

$$\left.\begin{aligned} \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\ \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b}; \end{aligned}\right\} \text{preaccumulation}$$

Fig. 2 An example for G^{*i}, with the nodes for the partials marked by ● and the nodes for the preaccumulation marked by ●.

variables, the edge labels of the remainder graph are required. In the example in Fig. 2, the required set is the maximal nodes $\{\frac{\partial q}{\partial a}, \frac{\partial q}{\partial b}\}$, but not node q, even though it too is maximal. The question now is how the values for the required nodes are provided: by storing, by recomputation from the minimal edges, or by a mix of the two.

2 A use case for storing edge labels

We limit the scope of the recomputation to the respective G^{*i}, and if we decide to always recompute from the minimal vertices, we replicate the TBR behavior. However, this may not be optimal, and in some cases one may prefer to store preaccumulated partials. Consider the computation of a coupled model $(q', p') = f(q, p)$, where we consider the q part of the model state for differentiation, and the coupling is done as a one-way forcing, such that $q' = f_1(q, p)$ and $p' = f_2(p)$, leaving the p portion of the model state passive. The scenario is illustrated in Fig. 3. Recomputing f would require the whole state (q, p), while propagating the adjoint values requires only the scarcity-preserving remainder graph edges. The original TBR analysis would store at least the portions of p that impact f_1 nonlinearly. Here we have not even considered the cost of (re)evaluating f_1 and f_2. If they are particularly expensive, then one may prefer to store edge labels or certain intermediate values as a tradeoff for the evaluation cost.

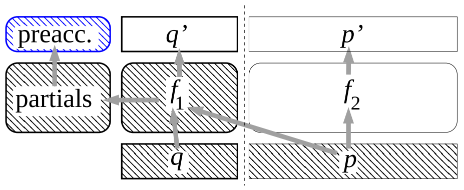


Fig. 3 Use case illustration, the shaded areas make up G^*.

3 Computational graphs share value restoration

For storing the required values, we can follow the TBR approach [8] by storing the values before they are overwritten. This information can be expressed as a bipartite graph $G_b = (\bigcup V^i_{min}, O, E_b)$. An example for G_b associated with two computational graphs G^1 and G^2 is given in Fig. 4. In the example, one can see that recovering the value for node a requires restores in overwrite locations o_1 and o_4; that implies that the value for node d is restored, and hence the value restoration benefits both graphs. Multiple overwrite locations for a given use are caused by aliasing, for example via the use of array indices or the use of pointers or branches in the control flow. The overwrite locations $o_k \in O$ can be vertices in $\bigcup(V^i_{inter} \cup V^i_{max})$ or "placeholder" vertices for variables (unique for each program variable) that go out of scope without the value in question being overwritten by an actual statement. That association is essential only for the final code generation and not for the formulation of the combinatorial problem.

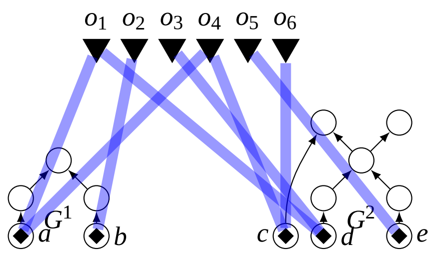


Fig. 4 An example for the graph G_b with respect to two computational graphs G^1 and G^2. The two node sets for G_b are shown as ⧫ and ▼ symbols, respectively.

4 Problem Formulation

We assume a set $\mathscr{G} = \{G^{*i}\}$ of extended computational graphs G^{*i}, as introduced in Sect. 1, along with their required sets $\mathscr{R}^i$ and one common bipartite use-overwrite graph G_b as introduced in Sect. 3. For each G^i, there is a bipartite use-overwrite subgraph $G_b[V^i_{min}] = G^i_b = (V^i_{min}, O^i, E^i_b)$ containing only the edges and vertices adjacent to V^i_{min}. The goal is to determine $S \subseteq O$ and $U \subseteq \bigcup V^i$ such that we minimize a static estimate for the number of values to be stored on tape.

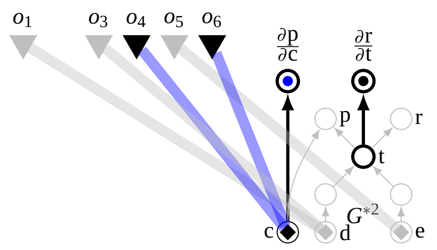
Given an S and U of values to be restored, we need to be able to recompute values in the remaining subgraph of G^{*i} such that all required nodes in $\mathscr{R}^i$ can be recomputed. To impose this as a formal condition on S and U, we denote with $G^*_{\mathscr{R}}$ the subgraph of G^* induced by all the nodes preceding $\mathscr{R}$.

Condition for Recomputation (CR): The sets S and U are sufficient to allow recomputation of nodes in all $\mathscr{R}_i$ if $\forall G^{*i}_{\mathscr{R}_i} : \exists$ vertex cut C^i with respect to $\mathscr{R}_i$ such that $\forall c_j \in C^i : (c_j \in U) \vee \big((c_j \in V^i_{min}) \wedge ((c_j, o) \in O \Rightarrow o \in S)\big)$.

In other words, if we know the values of all the vertices in the vertex cut C^i we are guaranteed to be able to recompute the values of the nodes in $\mathscr{R}_i$ by re-executing the computational graph between C^i and $\mathscr{R}_i$. A vertex in any of the cuts is either in U or it is not in U, in which case it must be a minimal vertex; if there is an overwrite of that value, then that overwrite location must be in S.

Consider the example shown in Fig. 5 where we reuse G^2 from Fig. 4 but add some example code for it and accordingly extend G^2 to G^{*2}. Note that for scarcity

```
t=5*d+4*e;
p=sin(c)+t;
r=cos(t);
```


o_1
o_3
o_4
o_5
o_6
$\frac{\partial p}{\partial c}$
$\frac{\partial r}{\partial t}$
p
r
t
G^{*2}
c
d
e

$$\frac{\partial t}{\partial d} = 4*1;$$
$$\frac{\partial t}{\partial d} = 5*1;$$
$$\frac{\partial p}{\partial t} = 1;$$
$$\frac{\partial r}{\partial t} = -\sin(t);$$
$$\frac{\partial p}{\partial c} = \cos(c);$$

Fig. 5 An example code (left) for the graph G^{*2} (center) highlighted for the subgraph with required nodes $\frac{\partial r}{\partial t}$ and $\frac{\partial p}{\partial c}$ that are computed from nodes $\{c, t\}$ according to the partials listing (right).

preservation we stop here with an incomplete elimination sequence [9] such that we have five remaining edges in the graph, of which only two are non-constant. These two are easily computable from nodes $\{c, t\}$ representing our vertex cut for the sub graph and for which values are to be restored. Therefore, we can choose $S = \{o_4, o_6\}$ and $U = \{t\}$. This choice emulates the behavior of TBR [8], whereas choosing $U = \{\frac{\partial r}{\partial t}\}$, for example, would be outside of the options TBR considers. A simple example for the benefits of a non-TBR choice is a sequence of assignments $v_i = \phi_i(v_{i-1}), i = 1, \ldots, n$ with non-linear ϕ_i for which one would prefer storing the single preaccumulated scalar $\frac{\partial v_n}{\partial v_0} = \prod_i \frac{\partial v_i}{\partial v_{i-1}}$ over the TBR choice of storing all arguments $v_i, i = 0, \dots, n-1$. On the other hand, for the example in Fig. 5, adding $\frac{\partial \mathrm{p}}{\partial \mathrm{c}}$ to U in exchange for $S = \emptyset$ would prevent any shared benefits that restoring node c has on restoring node a in G^1 as shown in Fig. 4.

4.1 A Cost Function

Because the decision about S and U has to be made during the transformation (aka compile time), any estimate regarding the runtime tape size cannot be more than a coarse indicator. Most significantly, the problem formulation disregards control flow, which may lead to an overestimate in case of branches containing the overwrite location or an underestimate if the overwrite location happens to be in a loop. On the other hand, the problem formulation as is allows for a very simple formulation of the cost function as $|S| + |U|$.

4.2 A Search Strategy

Because the choice of S impacts multiple graphs in $\mathscr{G}$ and thereby their contributions to U, there is no obvious best choice for a single $G^{*i}_{\mathscr{R}_i}$ that necessarily implies an optimal choice for all the other reduced combined graphs in general. For all but the simplest cases, the size of the search space implies that an exhaustive search is not a practical option. Therefore, we need a heuristic search strategy, and this strategy is crucial to obtain useful practical results from the proposed problem formulation.

One difficulty in devising a heuristic search strategy stems from the fact that on the one hand changing U or S are the elementary steps but on the other hand we have to adjust S or U, respectively, to satisfy (CR) so we may compute a valid cost function value on a consistent pair S, U. Adapting the sets to satisfy (CR) involves the determination of vertex cuts and therefore is rather costly. In addition to determining vertex cuts, one also has to satisfy that all the overwrite locations of the minimal vertices in the respective cuts are in S.

Therefore, it appears plausible to choose a search strategy that adds or removes elements in S in small but consistent groups. The two important special cases establishing upper bounds for the cost function are: (i) TBR, i.e. $U = \emptyset$ and S determined according to (CR), and (ii) saving preaccumulated partials, i.e. $U = \bigcup \mathscr{R}_i, S = \emptyset$. We pick the one with the lower cost and a given pair (S, U) as a starting point. One has to note at this point that case (i) is the original TBR case as presented in [8] only if the graphs G^i each represent one individual assignment statement. As soon as multiple assignments are flattened into a single graph G^i, the computed cost for case (i) will be less than or equal to that of the original TBR.

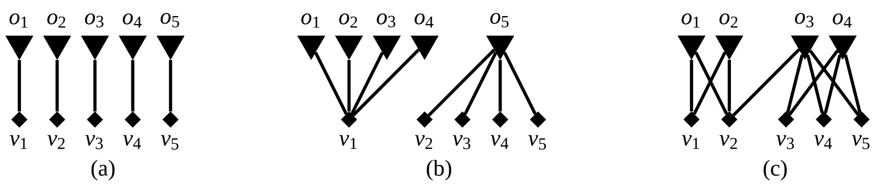


Fig. 6 Example scenarios for G_b.

While removing or adding the elements of S, we aim at making a change that will plausibly have the desired effect on the cost function. To get an indication of the effect on the cost function, we may limit our consideration to G_b. The most obvious observations for different scenarios in G_b which inform changes to S are shown in Fig. 6. It is clear that for (a) no particular preference of $v_i \in U$ vs. $o_i \in S$ can be deduced while for (b) $U = \{v_1\}$ and $S = \{o_5\}$ are preferred.

To make consistent changes to S we consider maximal bicliques covering G_b. For the moment, lets assume the bicliques are not overlapping, i.e. do not share nodes in V or O, as, for example, in Fig. 6 (a) and (b). For each biclique $B = (V_B, O_B)$, we can evaluate the node ratios for removing and adding to S

$$r_B^- = \frac{|O_B^-|}{|v_B|}; \quad r_B^+ = \frac{|v_B|}{|O_B^+|} \tag{3}$$

where $O_B^- = O_B \cap S$ and $O_B^+ = O_B \setminus S$. Obviously, the r^+ is only meaningful for $O_B^+ \neq \emptyset$ and otherwise we set r^+ to 0. A biclique $B^* = (v_B^*, o_B^*)$ with the maximal ratio has the potential for the largest impact on the cost function when applied as follows

$$S := \begin{cases} S \cup O_B^- & \text{if maximal ration is } r_B^+ \\ S \setminus O_B^- & \text{if maximal ratio is } r_B^- \end{cases} \tag{4}$$

If $S = \emptyset$, then all bicliques in Fig. 6(a) have ratio 1. In (b) the biclique for v_1 has $r^- = 0$ and $r^+ = 1/4$ while the one for o_5 has $r^- = 1/4$ and $r^+ = 4$. So, we start by adding o_5 to S as our first step. Clearly, in this setup only ratios greater than one are hinting at an improvement of the cost function. Ratios equal to one are expected to be neutral and those less than one are expected to be counterproductive.

After updating S, we apply (CR) to determine U and evaluate the cost function, compare to the minimum found so far, and accept or reject the step. If the step is rejected, we mark the biclique as rejected, remove it from further consideration for changes to S, restore the previous S, take the next biclique from the list order by ratio, and so on. If we accept the step, we mark the biclique as accepted and removed it from further consideration for changes to S.

Before we formalize the search algorithm, we have to address the case of overlapping bicliques as illustrated in Fig. 6(c). There, biclique $(\{v_2\}, \{o_3\})$ overlaps with $(\{v_1, v_2\}.\{o_1, o_2\})$ and $(\{v_3, v_4, v_5\}, \{o_3, o_4\})$. If we consider a biclique (V_B, O_B) with an overlap to another biclique in the V_B, then we need to add to O_B all nodes connected to the nodes in the overlap to obtain a consistent change to S. In our ex-

Algorithm 1 Apply (CR) an single combined,reduced DAG to update U

```
Given $G^*_{\mathscr{R}} = (V_{min} \cup V_{inter} \cup V_{max}, E), \mathscr{R}, S, V_S, U$
01 $U := U \setminus V$;    $C := V_S \cap V_{min}$
02 form the subgraph $G^{*'}$ induced by all paths from $V_m in \setminus C$ to $\mathscr{R}$
03 determine a minimal vertex cut $C'$ in $G^{*'}$ using as tie breaker the minimal distance from $C'$ to $\mathscr{R}$.
04 set $C := C \cup C'$ as the vertex cut for $G^*$ and set $U := U \cup C'$.
```

ample, this means that $(\{v_2\},\{o_3\})$ is augmented to $(\{v_2\},\{o_1,o_2,o_3\})$. After the augmentation, we no longer have a biclique cover and one may question whether starting with a biclique cover is appropriate to begin with. However, the rationale for starting with the minimal biclique cover (maximal bicliques) is to identify large groups of program variables whose overwrite locations are shared and for whom the store on overwrite yields a benefit to multiple uses. At the same time, using the minimal biclique cover implies a plausible reduction of the search space, compared to any other collection of bicliques which do not form a cover. While it is certainly possible to consider the case where one starts with a biclique collection that is not a cover, we currently have no rationale that prefers any such collection over the one where the V_B are the singletons $\{v_i\}$.

4.3 Algorithm

We formalize the method in Alg. 1 and Alg. 2. Assume from here on that $G_b = (V_b, O)$ is the subgraph induced by the vertices occurring in the $G^{*i}_{\mathscr{R}^i}$. For a given S, the subset of restored vertices $V_S \subseteq V_b$ contains the vertices whose successors are all in S. A choice of $\delta < 1$ permits a cut off in the search which disregards bicliques

Algorithm 2 Search algorithm for pair (S,U)

```
Given $\delta \in [0,1], \mathscr{R} = \bigcup \mathscr{R}^i, G^{*i}_{\mathscr{R}^i}$ for all $G^i \in \mathscr{G}$ and $G_b = (V_b, O, E_b)$; initialize $A = R = \emptyset$
01 if $|O| < |\mathscr{R}|$ then $(S,U) := (O,\emptyset)$;    $c := |O|$
02 else $(S,U) := (\emptyset,\mathscr{R})$;    $c := |\mathscr{R}|$
03 compute minimal biclique cover $\mathscr{C}$ for $G_b$
04 $\forall B = (V_B, O_B) \in \mathscr{C}$ set $O_B := O_B \cup \{o : ((v,o) \in E_b \wedge v \in V_B)\}$
05 while $\mathscr{C} \neq \emptyset$
06     $\forall B \in \mathscr{C}$ compute ratios $r^-_B$ and $r^+_B$ according to (3) and sort
07     if maximal ratio is less than $1-\delta$ exit with current $(S,U)$
08     update $S$ according to (4)
09     $\forall G^{*i}_{\mathscr{R}^i}$ update $U$ using Alg. 2
10     if $c \geq |S| + |U|$ then set $c := |S| + |U|$
11     else reset $S$ to the value it had before line 07
12 set $\mathscr{C} := \mathscr{C} \setminus \{B\}$
```

not expected to improve the cost function.

5 Observations and Summary

As pointed out in Sect. 4.1, the principal caveat to estimating a runtime memory cost by counting instructions (i.e. value overwrite locations or uses) as done here is the lack of control flow information. Conversely, for straight-line code, one will have either a single DAG graph when there is no aliasing or multiple DAGs with aliasing. In these cases, the algorithm presented here will produce a result better than or on par with the cases (i) and (ii). See Sect. 4.2, used as initialization in lines 01 and 02 of Alg. 2. The instruction count accurately reflects the runtime memory cost for a single execution of the straight-line code segment in question.

In the presence of control flow, the elements in U are correctly accounted for in the cost function by $|U|$ for a single execution of the DAG in which the respective vertices occur. In contrast, the runtime memory requirements for the elements in S are generally not related to the execution count of the DAGs for which the values are stored. It has been observed for the store-on-overwrite approach that the $|S|$ undercounts if it contains instructions in a loop and overcounts if its instructions are spread over mutually exclusive branches. Research related to the incorporation of the control flow is ongoing, but given the complexity of our flow-insensitive problem formulation clearly beyond the scope of this paper. Results that yield $U = \emptyset$ are on par or better than TBR (see 4.2). The problem formulation does not limit the number of DAGs in $\mathscr{G}$ to a single procedure, as long as the reaching definitions analysis that forms G_b is interprocedural. However, going beyond the scope of a single procedure increases the possibility of loop nesting and thus the error in the runtime cost estimate when Alg. 2 yields both S and U as non-empty.

While it is not the final answer to the general problem of storing versus recomputation, we view it as a stepping stone that widens the reach of automatic decisions by combining the information for multiple DAGs and permitting more recomputation through instructions flattened into DAGs. For a practical implementation one has to add logic excluding subgraphs of the combined graphs that evaluate to constant values and provide a means to recover the values for integer and address variables occuring in the memory references for storing and retrieving values from the tape at overwrite locations o to correctly match the memory references represented as the DAG vertices v through which the restored values are used for recomputations. This is already necessary for the original TBR algorithm. It is quite plausible to add expressions computing addresses or control flow conditions to the combined computational graphs and appropriately adding vertices to the set $\mathscr{R}$ of required

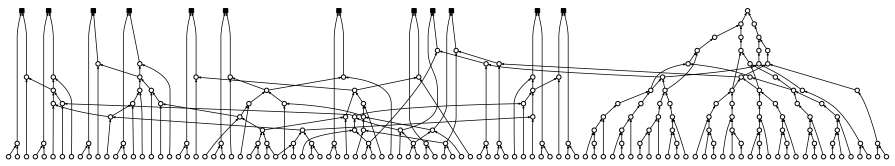


Fig. 7 Combined graph in OpenAD.

values so they become part of the automatic restore-recompute decisions. Then, the ordering for the adjoint code generation has to abide by certain dependencies of memory references in the vertices v upon addresses or indices also occuring as required values in the same graph. These are technical refinements that do not change the approach of the paper and are therefore left out. An implementation of the algorithms is forthcoming in OpenAD [11]. An example for G^* from a practical code using the experimental OpenAD implementation is shown in Fig. 7.

Acknowledgements This work was supported by the U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

1. Bischof, C.H., Bücker, H.M., Hovland, P.D., Naumann, U., Utke, J. (eds.): Advances in Automatic Differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 64. Springer, Berlin (2008). DOI 10.1007/978-3-540-68942-3
2. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York, NY (2005). DOI 10.1007/3-540-28438-9
3. Giering, R., Kaminski, T.: Recomputations in reverse mode AD. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) Automatic Differentiation: From Simulation to Optimization, Computer and Information Science, chap. 33, pp. 283–291. Springer, New York (2002). URL `http://www.springer.de/cgi-bin/search_book.pl?isbn=0-387-95305-1`
4. Giering, R., Kaminski, T.: Applying TAF to generate efficient derivative code of Fortran 77-95 programs. Proceedings in Applied Mathematics and Mechanics **2**(1), 54–57 (2003). URL `http://www3.interscience.wiley.com/cgi-bin/issuetoc?ID=104084257`
5. Griewank, A., Walther, A.: Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. ACM Transactions on Mathematical Software **26**(1), 19–45 (2000). URL `http://doi.acm.org/10.1145/347837.347846`. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.
6. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd edn. No. 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA (2008). URL `http://www.ec-securehost.com/SIAM/OT105.html`
7. Hascoët, L., Araya-Polo, M.: The adjoint data-flow analyses: Formalization, properties, and applications. In: Bücker et al. [2], pp. 135–146. DOI 10.1007/3-540-28438-9_12
8. Hascoët, L., Naumann, U., Pascual, V.: “To be recorded” analysis in reverse-mode automatic differentiation. Future Generation Computer Systems **21**(8), 1401–1417 (2005). DOI 10.1016/j.future.2004.11.009
9. Lyons, A., Utke, J.: On the practical exploitation of scarsity. In: Bischof et al. [1], pp. 103–114. DOI 10.1007/978-3-540-68942-3_10
10. Utke, J.: Flattening basic blocks. In: Bücker et al. [2], pp. 121–133. DOI 10.1007/3-540-28438-9_11
11. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. ACM Transactions on Mathematical Software **34**(4), 18:1–18:36 (2008). DOI 10.1145/1377596.1377598