RWTH Aachen

Department of Computer Science
Technical Report

On Optimal DAG Reversal

Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-05

RWTH Aachen · Department of Computer Science · March 2007

On Optimal DAG Reversal

Uwe Naumann

LuFG Informatik 12 (Software and Tools for Computational Engineering), Department of Computer Science
RWTH Aachen University, D-52056 Aachen Germany
WWW: `http://www.stce.rwth-aachen.de`, Email: `naumann@stce.rwth-aachen.de`

Abstract. Runs of numerical computer programs can be visualized as directed acyclic graphs (DAGs). We consider the problem of restoring the intermediate values computed by such a program (the vertices in the DAG) in reverse order for a given upper bound on the available memory. The minimization of the associated computational cost in terms of the number of performed arithmetic operations is shown to be NP-complete. The reversal of the data-flow finds application, for example, in the efficient evaluation of adjoint numerical programs. We derive special cases of numerical programs that require the intermediate values exactly in reverse order, thus establishing the NP-completeness of the optimal adjoint computation problem. Last but not least we review some state-of-the-art approaches to efficient data-flow reversal taken by existing software tools for automatic differentiation.

1 Motivation

The role of numerical simulation and optimization in computational science and engineering has gained significant importance over the last decades. Our ability to understand, for example, physical, chemical, and biological processes has improved with the growing computational resources and, more importantly, with the deepening insight into mathematical and algorithmic issues. Numerical simulation programs map potentially very large numbers of input parameters (let there be n) onto often much fewer outputs (say m of them, also referred to as the objectives). The classical numerical approach to quantifying the sensitivities of those objectives with respect to the parameters through finite difference quotients yields a computational complexity of $O(n)$. Note that certain high-end applications such as, for example, the simulation of ocean circulation (see, for example, `mitgcm.org`) may have a runtime of several days to produce physically relevant results on the latest high-performance computing platforms. The number of parameters may reach values of the order of $n = 10^9$. Hence, forward sensitivity analysis would require n runs of the simulation, which is simply not feasible.

Adjoint methods and corresponding program transformation techniques have been developed to replace the dependence on n with that on the number of objectives m. Often (for example, in least-squares approaches to data assimilation) the number of objectives is equal to one. In this case adjoint programs deliver the sensitivities of the objective with respect to all input parameters at $O(1)$.

Tangent-linear and adjoint codes can be generated from a given numerical simulation program by a semantic program transformation technique known as *automatic differentiation* (AD) [7]. A large number of successful applications of AD to real-world problems in science and engineering have been reported on in the proceedings of the four international conferences on the subject held in 1991 [4], 1996 [1], 2000 [3], and 2004 [2].

```
SUBROUTINE F(l, a, x, y)
  IMPLICIT NONE
  INTEGER l
  REAL a(2*l), x, y
  INTEGER i
  INTRINSIC SIN

  y = 1.
  DO i=1,l
    y = y * a(i)*SIN(a(l+i)*x)
  ENDDO
END
```

Fig. 1. Numerical Simulation Program

```
SUBROUTINE F_D(l, a, ad, x, y, yd)
  IMPLICIT NONE
  INTEGER l
  REAL a(2*l), ad(2*l), x, y, yd
  INTEGER i
  INTRINSIC SIN

  y = 1.
  yd = 0.0
  DO i=1,l
    yd = (yd*a(i)+y*ad(i))*SIN(a(l+i)*x) + y*a(i)*x*ad(l+i
      )*COS(a(l+i)*x)
    y = y*a(i)*SIN(a(l+i)*x)
  ENDDO
END
```

Fig. 2. Tangent-Linear Numerical Program (generated by Tapenade)

Example Consider the scalar function $y = F(\mathbf{a}, x)$ defined as

$$y = \sum_{i=0}^{2\cdot l-1} a_i \cdot \sin(a_{i+1} \cdot x) \quad .$$

An implementation in Fortran is shown in Figure 1. A centered-finite difference approximation of the gradient $F' = \frac{\partial y}{\partial \mathbf{a}}$ requires $2 \cdot n = 4 \cdot l$ function evaluations. The corresponding tangent-linear code $\dot{y} = \dot{F}(\mathbf{a}, \dot{\mathbf{a}}, x) \equiv F' \cdot \dot{\mathbf{a}}$ needs to be called n

times with $\dot{\mathbf{a}}$ ranging over the Cartesian basis vectors in $I\!R^n$. Refer to Figure 2 for a corresponding tangent-linear Fortran code generated by the AD tool Tapenade[1] [10]. The rules for generating tangent-linear code follow immediately from the chain rule of differentiation. The original code is augmented per assignment with the corresponding statement for computing a directional derivative of the left-hand side with respect to the active arguments (the components of $\mathbf{a}$ in our case) on the right-hand side. See line 11 in Figure 2.

```
 1 SUBROUTINE F_B(l, a, ab, x, y, yb)
 2   IMPLICIT NONE
 3   INTEGER l
 4   REAL a(2*l), ab(2*l), x, y, yb
 5   INTEGER i, ii1
 6   REAL temp, tempb
 7   INTRINSIC SIN
 8
 9   y = 1.
10   DO i=1,l
11     CALL PUSHREAL4(y)
12     y = y*a(i)*SIN(a(l+i)*x)
13   ENDDO
14   DO ii1=1,2*l
15     ab(ii1) = 0.0
16   ENDDO
17   DO i=l,1,-1
18     CALL POPREAL4(y)
19     temp = x*a(l+i)
20     tempb = SIN(temp)*yb
21     ab(i) = ab(i) + y*tempb
22     ab(l+i) = ab(l+i) + y*a(i)*COS(temp)*x*yb
23     yb = a(i)*tempb
24   ENDDO
25   yb = 0.0
26 END
```

Fig. 3. Adjoint Numerical Program (generated by Tapenade)

The adjoint code $\bar{\mathbf{a}} = \bar{F}(\mathbf{a}, x, \bar{y}) \equiv (F')^T \cdot \bar{y}$ shown in Figure 3 gives the gradient after a single run with $\bar{y} = 1$. For $n = 2 \cdot 10^4$ the centered finite difference approximation (similarly the tangent-linear code) takes roughly 45 seconds on our laptop. The adjoint code produces the same numerical result as the tangent-linear code in less than one second. Note that the adjoint code uses the repeatedly overwritten values of y (see line 12 in Figure 3) in reverse order. Tapenade inserts corresponding push and pop statements on lines 11 and 18 to store the required values on a stack. The rules for generating adjoint code derive from the

[1] See `http://tapenade.inria.fr:8080/tapenade/index.jsp`.

associativity of the chain rule taking into account overwrites in physical memory. Refer to [7] for further information on AD and its mathematical foundations.

The computational complexity of the gradient accumulation differs from that of the underlying function evaluation merely by a constant factor. The minimization of this factor is one of the major challenges in modern high-performance scientific computing. The difficulty arises from the fact that the data-flow in adjoint programs is reversed compared to that of the original simulation. Certain intermediate values computed by the simulation program need to be recovered in reverse order. Compare the used instances of y on line 12 with lines 21 and 22 in Figure 3. Overwriting of program variables and the fact that system memory is always limited (and way too small to store all intermediate values persistently) makes reaching this objective problematic. For example, the memory requirement of the adjoint code in Figure 3 will exceed the available memory for large values of l. The only solution lies in a hybrid approach that uses the available memory to store certain checkpoints from which other required values can be recomputed. Moreover, the recomputation should be as efficient as possible. The central question is the following: Which values should be stored and which ones should be recomputed such that the runtime of the adjoint code is minimized under the given constraints?

Optimizing the recomputation of an intermediate value amounts to playing the *pebble game* that is known to be PSPACE-complete [11]. Hence it is among the hardest problems that can be solved by a Turing machine using a polynomial amount of memory and unlimited time. Due to the missing time limit the distinction between deterministic and nondeterministic Turing machines is irrelevant in this case. Recall that the NP-complete problems are the hardest among those that can be solved by a nondeterministic Turing machine using a polynomial amount of memory and time. It is unknown whether the PSPACE-complete problems lie outside of NP or not. The same statement holds for NP and P, the class of problems that can be solved by a deterministic Turing machine using a polynomial amount of memory and time.

In this paper we assume persistent memory (main memory and/or files on hard disc for extremely large problems) for storing intermediate values. Recomputation is performed in nonpersistent memory (ideally in registers). The assumed nonpersistence derives from the maximum degree of freedom for memory access that is required for the intended optimization of the recomputation by heuristic approximation of a winning strategy for the pebble game. Hence, we consider a hierarchical setup where the recomputation part is optimized for a previously determined usage of the persistent memory. We assume unit cost for a STORE operation that writes to persistent memory as well as for a floating-point operation (FLOP) to compute a value as a function of the respective arguments. The cost of LOAD operations to access persistent or nonpersistent memory is assumed to be part of the following FLOP. Alternatively, one could argue that the LOAD cost is negligible due to *prefetching*. Moreover, we assume that the values to be made available in reverse order enter some nontrivial subsequent computation, for example, the computation of partial derivatives in the context of evaluating adjoints. These prerequisites represent a good approximation for our main target application, that is the efficient data-flow reversal in adjoint numerical programs.

The formalism is build on the representation of the program as a directed acyclic graph as described in Section 2. Existing algorithms are based on heuristics that use structural properties of the program (potentially augmented with conservative information on the computational costs of parts of the program) to derive a checkpointing scheme. Links to work in this area are given in Section 4. In this paper we formulate the OPTIMAL DATA-FLOW REVERSAL problem and we propose a proof for its NP-completeness. In Section 3 we establish the link with the OPTIMAL CHECKPOINTING problem in adjoint programs. To our knowledge the computational complexity of this combinatorial optimization problem has not been looked at so far.

2 Data-Flow Reversal

Following the notation in [7] we consider implementations of vector functions

$$F : I\!R^n \to I\!R^m, \quad \mathbf{y} = F(\mathbf{x}) \quad , \tag{1}$$

as computer programs that are expected to decompose for given inputs $\mathbf{x}$ into a single assignment code

$$v_j = \varphi_j(v_i)_{i \prec j} \quad , \tag{2}$$

for $j = 1, \ldots, q$. The notation $i \prec j$ is used to denote v_i as an argument of φ_j. There are n *independent* input variables $x_i = v_{i-n}$, $i = 1, \ldots, n$, p *intermediate* variables v_j, $j = 1, \ldots, p$, and m *dependent* output variables $y_k = v_{p+k}$, $k = 1, \ldots, m$. W.l.o.g., we assume that all dependent variables are mutually independent. We set $q = p + m$. A DAG $G = (V, E)$ is induced by the single assignment code. Vertices represent the single assignment code variables whereas edges encode the dependence relation as $(i, j) \in E$ if and only if $i \prec j$. Hence, G has n sources and m sinks.

The OPTIMAL DATA-FLOW REVERSAL problem is to recover the values of the $n + q$ single assignment code variables in reverse order for a given upper bound $n \leq K \leq n + q$ on the available memory and such that the number of required FLOPs becomes minimal. This number vanishes identically for $K = n + q$. In reality one rarely has that much memory at ones disposal. The values of the inputs need to be stored in any case as they cannot be recomputed from other values. Hence, $K \geq n$.

Example The single assignment code of the straight-line program[2]

$$\begin{aligned} t &= x_0 \cdot \sin(x_0 \cdot x_1) \\ x_0 &= \cos(t) \\ x_1 &= t/x_1 \end{aligned}$$

is shown in Figure 4. The objective is to recover $v_5, \ldots, v_{-1}$ in that order for the given upper bound K on the available memory. For $K = n + q = 7$ a store-all strategy uses the available memory like a stack. For $K = 2$ one stores v_{-1} and v_0 followed by recomputing v_5 (4 FLOPs), v_4 (4 FLOPs), v_3 (3 FLOPs), v_2 (2 FLOPs), and v_1 (1 FLOP) at a cumulative cost of 14 FLOPs. The number of

[2] The flow of control is fixed by evaluation of the program at a given input. Hence, we can focus on straight-line programs.

$v_{-1} = x_0; \quad v_0 = x_1$
$v_1 = v_{-1} \cdot v_0$
$v_2 = \sin(v_1)$
$v_3 = v_{-1} \cdot v_2$
$v_4 = \cos(v_3)$
$v_5 = v_3 / v_0$
$x_0 = v_4; \quad x_1 = v_5$

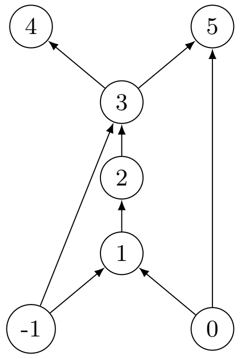

Fig. 4. Single Assignment Code and Corresponding DAG

intermediate and dependent single assignment code variables q is a sharp lower bound on the cost for making these values available. A single function evaluation is required in any case. There are data-flow reversals that have unit cost per recomputed value based on certain stored values. For example, in Figure 4 one could only store v_{-1}, v_0, v_2, and v_3. The values of v_5, v_4, and v_1 can be recomputed as a function of their stored predecessors at the cost of a single FLOP each. Hence, one can ask for a data-flow reversal with unit cost per recomputed value that consumes minimal memory. We refer to the corresponding decision problem as the Minimum Memory Data-Flow Reversal (MMDFR) problem.

In order to prove the NP-completeness of MMDFR we pick a known NP-complete problem P and we design a polynomial transformation from each instance of P to an instance of MMDFR. Furthermore, we need to show that there is a solution for the given instance of P if and only if there is one for its counterpart in MMDFR. We pick $P =$Vertex Cover.

Vertex Cover (VC) Given a graph $G = (V, E)$ is there a subset $W \subseteq V$ of size $\omega \leq \Omega$, s.th. each edge in E is incident with at least one vertex from W?

Theorem VC is NP-complete.

Proof See [5]. ■

Vertex Cover in Dags (VCD) Given a directed acyclic graph $G = (V, E)$ is there a subset $W \subseteq V$ of size $\omega \leq \Omega$, s.th. each edge in E is incident with at least one vertex from W, that is either the source or the target (or both) of each edge is in W?

Proof Consider an arbitrary instance of VC on a graph $G = (V, E)$. Enumerate all vertices and make edges directed s.th. $(i, j) \in E \Leftrightarrow i < j$. This procedure is illustrated by Figure 5 (a) and (b). The resulting directed graph G' has no cycles. Obviously, it has a vertex cover W' of size $\omega \leq \Omega$ if and only if G has a vertex cover W of the same size. Simply set $W' = W$. ■

Formally, the MMDFR problem is stated as follows.

MMDFR Given are a DAG G and an integer $n \leq K \leq n+q$. Is there a data-flow reversal with cost $n + q$ that uses $k \leq K$ memory?

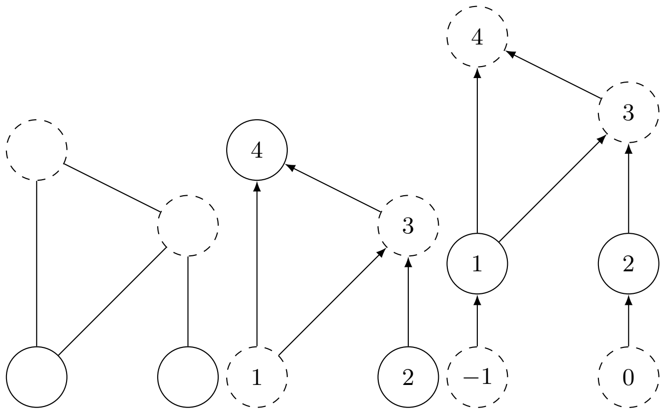

Fig. 5. Reduction VC → VCD → MMDFR

Before we prove MMDFR to be NP-complete we have a quick look at an example. Consider Figure 5 (c). For $K = 6$ store-all is a solution to MMDFR. For $K = 5$ we could store v_{-1} and v_0 in addition to any three variables out of $v_1, \ldots, v_4$. Storing v_1 and v_3 or v_3 and v_4 would solve MMDFR for $K = 4$. You may wish to verify that there is no solution for MMDFR with $K \leq 3$. Equivalently, there is no vertex cover of size $\omega \leq 1$ for the graph in Figure 5 (a) nor for its directed acyclic versions, one of which is shown in Figure 5 (b).

Theorem MMDFR is NP-complete.

Proof We reduce from VCD. A given solution of MMDFR is verified trivially by counting the number of FLOPs for persistent memory of a given size.

Derive an instance of MMDFR for any instance of VCD on a DAG $G' = (V', E')$ as follows: Add a unique predecessor to each of the n' sources in G' to get $G = (V, E)$. Hence, $V' = \{1, \ldots, q\}$ and $V = X \cup V'$, where $X = \{1-n, \ldots, 0\}$. Moreover $n = n'$ and $m = m'$. We claim that there is a solution for MMDFR on G with $K = \Omega + n$ if and only if there is a solution for VCD with Ω on G'.

"$\Leftarrow$" We need to store the n sources of G as they cannot be recomputed. For a given vertex cover $W \subseteq V'$ of G' we observe that the predecessors of any $v \in V' \setminus W$ are in W. Hence, the values in $V' \setminus W$ can be recomputed at unit cost from stored values. It follows that the overall cost for MMDFR is less than or equal to $K = \Omega + n$ if $|W| \leq \Omega$.

"$\Rightarrow$" Consider a solution for MMDFR with overall cost $n + q$ and memory requirement $k \leq \Omega + n$ for storing $M \subseteq V$. Suppose that $W \equiv M \setminus X$ is not a vertex cover in G'. Hence, there is an edge $(i, j) \in E'$ with $i \notin W$ and $j \notin W$. All values v_k, $k = 1, \ldots, p$, $k \neq j$ can be obtained at unit cost by either restoring their previously stored values (if $k \in W$) or recomputing them from known values at unit cost by a single FLOP (if $k \notin W$). The recomputation of v_j takes two FLOPs as it involves the recomputation of v_i whose value is nonpersistent and has therefore to be recomputed again at the cost of a single FLOP. The overall cost of the given MMDFR solution adds up to $n + q - 2 + 3 = n + q + 1$ which is no longer optimal (contradiction).

The reuse of already vacated persistent memory to store v_j does not change this situation. We still need to perform one FLOP to recompute v_j. The subsequent STORE has unit cost as well as the FLOP required to recompute v_i. The total costs still exceeds the optimum by one.

In general we observe quadratic growth in arithmetic complexity along any path through vertices that are not in W if previously vacated persistent memory is not reused. Otherwise we still get linear growth. The optimal cost is achieved only if all these paths have length zero or one implying that W must be a vertex cover in G'. ■

MMDFR is merely an intermediate step toward the problem that we are actually interested in. Instead of fixing the overall cost and minimizing the required memory we are really faced with an upper bound on the available memory. Our objective is to use this memory “wisely”, that is to store as many values as possible such that the overall cost becomes minimal. We refer to this combinatorial optimization problem as the FIXED MEMORY MINIMUM COST DATA-FLOW REVERSAL or simply the OPTIMAL DATA-FLOW REVERSAL (ODFR) problem. It can be formulated as a decision problem as follows.

ODFR Given are a DAG G and integers K and C such that $n \leq K \leq n + q$ and $K \leq C$. Is there a data-flow reversal that uses at most K memory and costs $c \leq C$?

Theorem ODFR is NP-complete.

Proof An algorithm for ODFR can be used to solve MMDFR as follows: For $K = n + q$ store-all is a solution of ODFR for $C = n + q$. Now decrease K by one at a time as long as there is a solution of ODFR for $C = n + q$. The smallest K for which such a solution exists is the solution of the minimization version of MMDFR. Again, a given solution is trivially verified in polynomial time by counting the number of FLOPs performed by the respective code. ■

3 Link with Adjoints

Recall from Section 1 that the adjoint version of a numerical program F as defined in Equation (1) computes adjoints $\bar{\mathbf{x}}$ of the inputs as a function of $\mathbf{x}$ and given adjoints $\bar{\mathbf{y}}$ of the outputs according to

$$\bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) \equiv F'(\mathbf{x})^T \cdot \bar{\mathbf{y}} \quad .$$

The OPTIMAL CHECKPOINTING problem for $\bar{F}$ is to determine for a given upper bound K on the available persistent memory a set of values computed by the single assignment code as defined in Equation (2) such that the computational cost of the adjoint propagation becomes minimal. In the following we distinguish between two variants of adjoint propagation, due to the incremental and nonincremental reverse modes of AD [7].

The link between ODFR and OPTIMAL CHECKPOINTING in adjoint computations is not immediately apparent. Referring back to the example in Section 1 we note that adjoint codes do not necessarily use the intermediate values in strictly reverse order. For example, the value of v_{-1} is used prior to that of v_1 as it labels edge $(2, 3)$ in Figure 4 whereas v_1 is an argument of the local partial derivative $\cos(v_1)$ that labels edge $(1, 2)$. In order to apply ODFR we need to show that there are numerical programs whose adjoints use the intermediate values in strictly reverse order. The order in which the intermediate values are used depends on the order of the computation of the local partial derivatives (the edge labels in the linearized DAG).

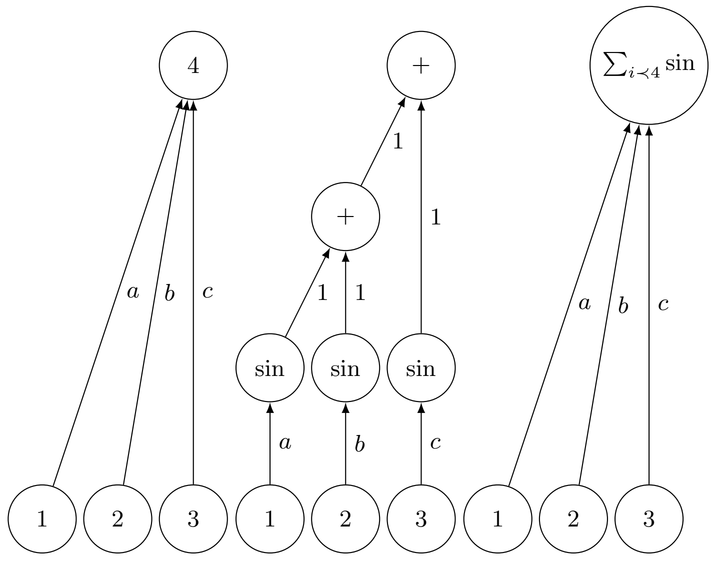

Fig. 6. Reduction DAG → Computational Graph (Nonincremental Adjoints)

3.1 Nonincremental Adjoints

In nonincremental reverse mode the labels of the outgoing edges are computed for all vertices in the linearized DAG (second line in Equation (3)).

$$\begin{aligned} v_j &= \varphi_j(v_i)_{i \prec j} \quad \text{for } j = 1, \ldots, q \\ c_{kj} = \frac{\partial \varphi_j}{\partial v_k} \quad \text{for } j \prec k \text{ and } \bar{v}_j &= \sum_{k:j \prec k} c_{kj} \cdot \bar{v}_k \quad \text{for } j = q, \ldots, 1-n \end{aligned} \quad . \tag{3}$$

In order to apply ODFR to nonincremental adjoints we construct a numerical program such that all edge-labels (local partial derivatives) depend only on the value of the respective edge's source. For $j = 1, \ldots, q$ set, for example,

$$v_j = \sum_{i \prec j} \sin(v_i) \quad \text{where} \quad \frac{\partial \sin}{\partial v_i} = \frac{\partial \sin}{\partial v_i}(v_i) \equiv \cos(v_i) \quad .$$

Constant folding according to $1 \cdot x = x$ gives the original graph [14]. The procedure is illustrated by Figure 6. The resulting nonincremental adjoint code requires all intermediate variables in reverse order, implying the ODFR problem.

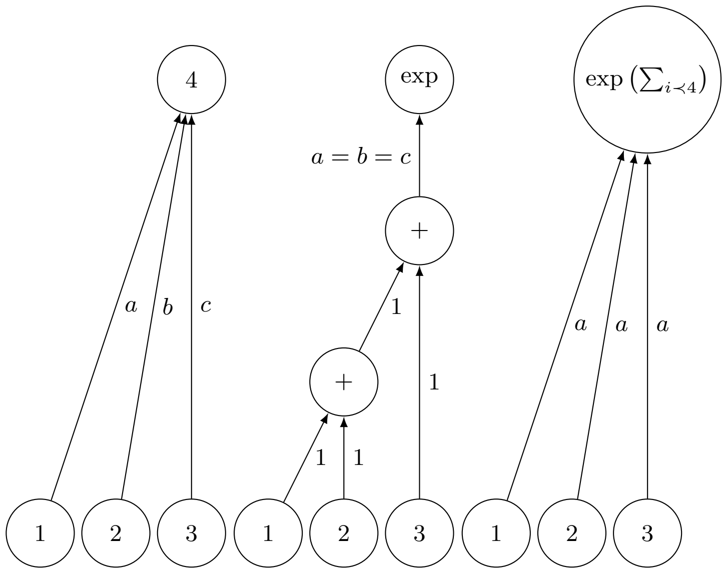

Fig. 7. Reduction DAG → Computational Graph (Incremental Adjoints)

3.2 Incremental Adjoints

In incremental reverse mode the labels of the incoming edges are computed for all vertices in the linearized DAG (second line in Equation (4)).

$$\begin{aligned} v_j &= \varphi_j(v_i)_{i \prec j};\ \bar{v}_j = 0 \quad \text{for } j = 1, \ldots, q \\ c_{j,i} &= \frac{\partial \varphi_j}{\partial v_i};\ \bar{v}_i = \bar{v}_i + c_{j,i} \cdot \bar{v}_j \quad \text{for } i \prec j \text{ and } j = q, \ldots, 1 \end{aligned} \quad . \tag{4}$$

All edge-labeled DAGs can be regarded as linearized computational graphs, such that the edge labels (local partial derivatives) depend only on the value of the respective edge's target. A constructive description of a corresponding numerical program is the following. For $j = 1, \ldots, q$ set

$$v_j = \exp\left(\sum_{i \prec j} v_i\right) \quad \text{such that} \quad \frac{\partial v_j}{\partial v_i} \equiv v_j \quad .$$

Again, constant folding according to $1 \cdot x = x$ gives the original graph as shown in Figure 7. As a result the incremental adjoint code requires all intermediate variables in reverse order, implying the ODFR problem.

4 Approaches to Efficient Data-Flow Reversal in Automatic Differentiation

Reversal of the data flow in numerical programs yields the need for control-flow reversal. Various methods have been investigated in the literature covering both intraprocedural [17] and interprocedural flow of control [6, 7, 15]. Program analysis plays a crucial role in adjoint code generation [8, 9]. As a compile-time activity the code generation needs to be conservative in the sense that correct adjoints are guaranteed to be computed for arbitrary inputs. Hence, adjoint compilers should always put robustness above efficiency. Nevertheless efficiency is often crucial, especially in the context of large-scale numerical simulations where a factor between the runtimes of the original and the adjoint codes of six and more may already be too large for the adjoint mode to be applicable. An example for such an application is the MIT general circulation model [16].

Relevant special cases of the OPTIMAL CHECKPOINTING problem have been considered in [7, 18, 19]. The authors focus on time evolutions

$$F : \mathbb{R}^n \to \mathbb{R}^n$$

with

$$\mathbf{x}_l = F(\mathbf{x}_0) = f_l(f_{l-1}(\dots(f_1(\mathbf{x}_0))\dots)) \tag{5}$$

and $\mathbf{x}_i = f_i(\mathbf{x}_{i-1})$ for $i = 1, \dots, l$ that are implemented as loops updating the state vector $\mathbf{x}$ at each iteration i. Given values for l and s (number of checkpoints, that is copies of the state vector, that can be stored persistently), the aim is to solve the EVOLUTION REVERSAL (ER) problem that is to minimize the time $t(l, s)$ needed for the reversal of F. For non-uniform step costs t_i for $i = 1, \dots, l$ one gets

$$t(l, s) = \min_{1 \le \hat{l} < l} \left(\sum_{i=1}^{\hat{l}} t_i + t(l - \hat{l}, s - 1) + t(\hat{l}, s) \right)$$

The elements of the right-hand side of this relation are

- $\sum_{i=1}^{\hat{l}} t_i$ – the cost for advancing to the checkpoint at $\hat{l}$;
- $t(l - \hat{l}, s - 1)$ – the cost for reversing the right subchain with $s - 1$ checkpoints;
- $t(\hat{l}, s)$ – the cost for reversing the left subchain.

The number of potential reversal schemes is exponential in l. Fortunately, the following two properties qualify the ER problem for dynamic programming. *Overlapping subproblems:* The problem for l includes the problems for $\hat{l} \le l$. *Optimal substructure:* A solution for l is optimal if and only if it solves all its subproblems.

We consider the work on efficient / optimal loop reversal as a special case of the OPTIMAL CHECKPOINTING problem to be highly relevant. Obviously, loops are the main reason for numerical programs becoming large-scale in terms of computational complexity. Solutions for the ER problem and its variants are key ingredients of efficient adjoint computations. The proof that OPTIMAL CHECKPOINTING is in fact NP-complete can be seen as justification for past and ongoing work on relevant special cases as well as on heuristics for approximately solving the general problem.

5 Summary and Conclusion

In this paper we have considered the optimal reversal of directed acyclic graphs of numerical simulation programs. Given was an upper bound on the persistent memory available for storing the values associated with the vertices in the graph. These values needed to be recovered in reverse order while keeping the computational complexity to a minimum. The corresponding OPTIMAL DATA-FLOW REVERSAL problem has been shown to be NP-complete. The NP-completeness of the OPTIMAL CHECKPOINTING problem in adjoint computations followed.

Our results should be regarded as contributions to a better understanding of the theoretical foundations of adjoint computations. The need for efficient adjoint codes is likely to increase with the ongoing progress in work on large-scale inverse problems and nonlinear optimization. The potential impact of the idea behind the proof on algorithm development remains unclear. Polynomial reductions to other well-studied NP-complete problems may grant us access to a variety of approximation algorithms and / or powerful heuristics. Novel approaches to adjoint code generation will most likely be based on static information on the code structure, possibly augmented with dynamic information about the computational complexities of code fragments generated by profiling runs. The reversal of address computations represents another very important factor. Work is underway to formalize the address computation reversal problem and to provide algorithms for its (approximate) solution.

We would like to conclude with an important observation. The automatic generation of efficient adjoint code must be regarded as one of the major challenges in compiler-based semantic transformation of numerical programs. It requires expertise in numerical analysis as well as compiler construction. A large number of highly interesting combinatorial problems arise. The ability to generate adjoint code automatically will become an important feature of special compilers for scientific computing. Work is underway to develop prototypes to demonstrate the usefulness of this approach [12, 13]. Development and support of an industrial-strength adjoint code compiler remains a major task in software engineering.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
2. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2005. Springer.
3. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
4. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.
5. M. Garey and D. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.
6. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.
7. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* SIAM, Apr. 2000.
8. L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In *[2]*, pages 135–146. Springer, 2005.
9. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
10. L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.
11. R. Tarjan J. Gilbert, T. Lengauer. The pebbling problem is complete for polynomial space. *SIAM J. Comput.*, (9):513–524, 1980.
12. M. Maier and U. Naumann. Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In G. Montero B.H.V. Topping and R. Montenegro, editors, *Proceedings of the Fifth International Conference on Engineering Computational Technology.* Civil-Comp Press, Kippen, Stirlingshire, United Kingdom, 2006.
13. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In *[2]*, pages 159–170. Springer, 2005.

14. U. Naumann and J. Utke. Optimality-preserving elimination of linearities in Jacobian accumulation. *Electronic Transactions on Numerical Analysis*, 21:134–150, 2005. Special Volume on Combinatorial Scientific Computing.
15. U. Naumann and J. Utke. Source templates for the automatic generation of adjoint code through static call graph reversal. In P. Sloot et al., editor, *Computational Science - ICCS 2005, Proceedings of the International Conference on Computational Science, Atlanta, GA, USA, May 22-25, 2005, Part I*, volume 3514 of *LNCS*, pages 338–346. Springer, 2005.
16. P. Heimbach and C. Hill and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
17. J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the intraprocedural flow of control in adjoint computations. *Journal of Systems and Software*, 79:1280–1294, 2006.
18. A. Walther. *Program Reversal Schedules for Single- and Multi-processor Machines*. PhD thesis, Institute of Scientific Computing, Technical University Dresden, Germany, 1999.
19. A. Walther and A. Griewank. New results on program reversals. In *[3]*, chapter 28, pages 237–243. Springer, 2001.