Exploitation of Structural Sparsity in Algorithmic Differentiation

Von der Fakultät Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Ebadollah Varnik

aus Bandar Torkman, Iran

Berichter : Universitätsprofessor Dr. Uwe Naumann
Universitätsprofessor Dr. Andrea Walther

Tag der mündlichen Prüfung : 09.11.2011

Contents

List of Symbols

Abstract

The background of this thesis is algorithmic differentiation (AD) [GW08] of in practice very computationally expensive vector functions $F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m$ given as computer programs. Traditionally, most AD software[1] provide forward and reverse modes of AD for calculating the Jacobian matrix $\nabla F(\mathbf{x})$ accurately at a given point $\mathbf{x}$ on some kind of internal representation of F kept on memory or hard disk. In fact, the storage is known to be the bottleneck of AD to handle larger problems efficiently in reverse mode. For instance, a tape is the internal representation of choice in the C++ operator overloading tool ADOL-C [GJM$^+$99] that presents an augmented version of F. Thus, ∇F can be obtained in forward and reverse fashion by an interpretative forward and reverse propagation of directional derivatives and adjoints [NMRC07] through the tape, respectively. The forward mode AD can be implemented very cheaply in terms of memory by single forward propagation of directional derivatives at runtime (tapeless in ADOL-C terminology). However, the reverse mode needs to store some data [HNP05] in the so-called forward sweep to allow the data flow reversal [Nau08] needed for backward propagation of adjoints. The latter is recently the focus of ongoing research activities of the AD community for $m = 1$ as a single application of reverse mode is enough to accumulate the gradient of F. To handle the memory bottleneck, checkpointing schedules e.g. revolve [GW00] have been developed for time-dependent problems. However, they require user's knowledge in both the function F as well as the reverse mode AD. In this context, we aim to provide a tool, which **minimizes non-AD experts effort** in application of the reverse mode AD on their problems for large dimensions.

Chapter 2 of this thesis is concerned with the accumulation of the Jacobian of F by the application of elimination techniques, which are very close to the Gaussian elimination performed in sparse LU factorization [PT08, FTPR04]. Thereby, we present algorithms that allow the application of elimination techniques [GN02] to the very large and sparse extended Jacobian of F being a lower triangular matrix of *local partial derivative*. However, the extended Jacobian is of quadratic memory complexity. Hence, compressed row storage [DER86] (CRS) representation is used to exploit its sparsity. This is done by first performing the so-called symbolic elimination step on the corresponding *bit pattern* of the extended Jacobian. This step predicts storage required for the statically allocated target CRS, which is used to accumulate the Jacobian of F at $\mathbf{x}$.

Nonetheless, the capability of the static CRS is also bounded by the memory consumption of the respective bit pattern even though the memory usage of CRS is considerably lower. To tackle this problem, elimination techniques are applied locally to the dense extended Jacobian (i.e without exploiting sparsity) and its CRS representation. Therefore, we keep track of the memory usage during the evaluation of F and apply elimination techniques whenever the memory bound is reached. The elimination is supposed to free memory enabling us to continue evaluating F. In fact, the evaluation of ∇F may require multiple evaluation and elimination steps. The former is supposed to provide the target data structure on which the latter is performed. We refer to this approach as *iterative Jacobian accumulation*.

The implementations of the ideas above are provided in the C++ operator overloading tool DALG[2]

[1] Existing AD tools can be found on the community website `www.autodiff.org`.

[2] DALG stands for Derivative Accumulation for Large Graphs.

attached to this work. DALG can be used to accumulate Jacobians and gradients very cheaply in terms of memory automatically without any user intervention and knowledge about the underlying function.

Moreover, in Chapter 3 we investigate methods to improve the exploitation of structural sparsity of in general derivative tensors such as Jacobians and Hessians. Existing methods are based on the knowledge of the nonzero pattern of target derivative structures, where a compression is usually achieved by the application of some coloring algorithms [GMP05] to a graphical representation. We consider partial distance-2 coloring and star/acyclic coloring of the bipartite and adjacency graph of Jacobians and Hessians, respectively, provided by the coloring package ColPack [NNH$^+$11]. Hence, whenever we talk about coloring Jacobians and Hessians we mean the coloring of the respective graphs.

To achieve better compression, we distinguish between **variable** and **constant nonzeros**, where the latter is supposed to be unchanged at all those points of interest with fix flow of control. Hence, only the former is needed to be computed at runtime. Therefore, general runtime algorithms are provided to compute the variable pattern and the constant entries. We test also their performance in both runtime and achieved colors in the process of sparse Jacobian and Hessian computation.

Furthermore, we present an algorithm to overestimate the Hessian sparsity pattern that is referred to as the **conservative Hessian pattern estimation**. It is the result of exploiting the partial separability of F. We present numerical results on the computational cost as well as the coloring performance in terms of runtime and achieved colors of the conservative pattern and compare them with those of the exact (nonzero) Hessian pattern. The computational complexity of the latter is known to be quadratic as proposed by Walther [Wal08].

Finally, the conservative algorithm is refined to a recursive version that is referred to as the **recursive Hessian pattern estimation**. The recursive algorithm is supposed to converge to the exact one in both runtime and the resulting pattern for sufficiently large recursion level. Thereby, the recursion level one yields exactly the same pattern as the conservative one.

Acknowledgment

At first place, I wish to thank to my supervisor, Prof. Dr. Uwe Naumann for his guidance and support throughout this work. Special thanks go also to Prof. Dr. Andrea Walther at the University of Paderborn who has supported me with test cases suitable for this thesis.

Moreover, I would like to thank to my colleges Lukas Razik, Viktor Mosenkis, Ekkapot Charoenwanit, Markus Beckers, and Johannes Lotz at Lehr- und Forschungsgebiet Informatik 12, Software and Tools for Computational Engineering (STCE) of the RWTH Aachen University who have supported me in any respect during the completion of the thesis. I wish to thank also the rest of the STCE group, for all the discussions that have considerably broadened my knowledge in computer science.

Special thanks go also to my colleges Georg Schramm and Sascha Bücken at Computing and Communication Center of RWTH Aachen University who have supported me with the test system used in this work.

Finally, I am grateful to my family, especially my Wife Olga and my children Fabian and Olesja for their support through this phase of my life.

Chapter 1

Foundations

The main focus of this thesis is *algorithmic differentiation* (AD) [CG91, BBCG96, CFG+02, BCH+06, BBH+08] of multivariate vector functions

$$F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}) \tag{1.1}$$

mapping a vector of inputs $\mathbf{x} \in \mathbb{R}^n$ onto a vector of outputs $\mathbf{y} \in \mathbb{R}^m$.

Assumption 1.1. *F is d times continuously differentiable in some neighborhood of the given argument $\mathbf{x} \in D$ for a given derivative degree $d \geq 1$.*

Furthermore, F is assumed to be implemented in some high-level imperative programming language like C, C++ or Fortran. Whenever we talk about F we mean the corresponding implementation that is assumed to decompose into a *single assignment code* (SAC) at every point of interest as

$$\begin{aligned} &\text{for } j = n+1, \ldots, q \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \end{aligned} \tag{1.2}$$

with $q = n + p + m$ and $i \prec j$ denoting a direct dependence of v_j on v_i. The transitive closure of this relation is denoted by $\prec^*$. Thereby, the result of each *elemental function*

$$\varphi_j : \quad \mathbb{R}^l \supseteq D_{\varphi_j} \to \mathbb{R}$$

is assigned to a unique auxiliary variable v_j with $l = |P_j|$. By P_j [S_j] we denote the set of indices of all arguments of φ_j [those variables that have v_j as argument]. Obviously, the basic arithmetic operations $\{+, -, *, \backslash\}$ as well as the elementary functions $\{\sin, \cos, \tan, \exp\}$ provided by the most imperative programming languages are elemental. Thereby, we observe that the number of arguments of most intrinsics is bounded by two [Nau99]. The n *independent inputs* $\mathbf{x} \equiv (v_i)_{i=1,\ldots,n}$ are mapped onto m *dependent outputs* $\mathbf{y} \equiv (v_{n+p+j})_{j=1,\ldots,m}$ involving the computation of the values of p *intermediate variables* $\mathbf{z} \equiv (v_{n+k})_{k=1,\ldots,p}$. Moreover, all those variables in SAC that represent current instances of program variables are referred to as *alive* variables, otherwise they denote *dead* ones.

1.1 First and Second-Order Derivative Models

In the following we use the notation of *The Art of Differentiating Computer Programs* by Naumann [Nau11]. We introduce here first and second order AD models that are used in this work.

The *tangent-linear model* (TLM)

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv < \nabla F(\mathbf{x}), \mathbf{x}^{(1)} >= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \tag{1.3}$$

of F computes $\mathbf{y}^{(1)} \in \mathbb{R}^m$ as the product of the Jacobian defined by Equation (1.7) times a direction vector $\mathbf{x}^{(1)} \in \mathbb{R}^n$, where the expression

$$< A, \mathbf{u} >= \mathbf{b} \quad \text{with} \quad \mathbf{b} \equiv (b_j)_{j=1,\ldots,m} \tag{1.4}$$

represents a *tangent-linear projection* of the matrix $A \in \mathbb{R}^{m \times n}$ in direction $\mathbf{u} \in \mathbb{R}^n$ with $b_j =< a_{j,*}, \mathbf{u} >= \sum_{i=1}^{n} a_{j,i} \cdot u_i$ denoting the usual inner product of two vectors in $\mathbb{R}^n$ for $j = 1, \ldots, m$.

The *adjoint model* (ADM)

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv < \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) >= \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \tag{1.5}$$

of F computes $\mathbf{x}_{(1)} \in \mathbb{R}^n$ as the product of the transposed Jacobian $\nabla F(\mathbf{x})^T$ times an adjoint vector $\mathbf{y}_{(1)} \in \mathbb{R}^m$, where the expression

$$< \mathbf{w}, A >= \mathbf{c} \quad \text{with} \quad \mathbf{c} \equiv (c_i)_{i=1,\ldots,n} \tag{1.6}$$

represents an *adjoint projection* of $A \in \mathbb{R}^{m \times n}$ in direction $\mathbf{w} \in \mathbb{R}^m$ with $c_i =< a_{*,i}, \mathbf{w} >= \sum_{j=1}^{m} a_{j,i} \cdot w_j$ denoting the usual inner product of two vectors in $\mathbb{R}^m$ for $i = 1, \ldots, n$. We note here that TLM and ADM are also known as forward and reverse mode AD models, respectively.

Thus, the Jacobian

$$(\mathbb{R}^{m \times n} \ni) \ \nabla F = \nabla F(\mathbf{x}) \equiv \left(f'_{j,i}\right)_{i=1,\ldots,n}^{j=1,\ldots,m} \tag{1.7}$$

of F being the matrix of the first-order partial derivatives also referred to as *sensitivities* $f'_{j,i} = \frac{\partial y_j}{\partial x_i}(\mathbf{x})$ of F at point $\mathbf{x} \in R^n$ can be accumulated using $F^{(1)}$ [$F_{(1)}$] by letting $\mathbf{x}^{(1)}$ [$\mathbf{y}_{(1)}$] range over Cartesian basis vectors of the input [output] spaces $\mathbb{R}^n$ [$\mathbb{R}^m$]. Hence, accumulating $\nabla F(\mathbf{x})$ using TLM and ADM can be done at the computational cost of $O(n) \cdot Cost(F)$ and $O(m) \cdot Cost(F)$, respectively, where $Cost(F)$ denotes the computational cost of evaluating F. In case of $m = 1$, the gradient of F can be accumulated very cheaply in terms of runtime by single evaluation of the ADM of F.

The corresponding vector formulation of both models above are referred to as *tangent-linear vector model* (TLVM)

$$Y^{(1)} = \nabla F(\mathbf{x}) \cdot X^{(1)} \tag{1.8}$$

and *adjoint vector model* (ADVM)

$$X_{(1)} = \nabla F(\mathbf{x})^T \cdot Y_{(1)} \quad , \tag{1.9}$$

where $X^{(1)} \in \mathbb{R}^{n \times k}$, $Y^{(1)} \in \mathbb{R}^{m \times k}$ and $X_{(1)} \in \mathbb{R}^{n \times k}$, $Y_{(1)} \in \mathbb{R}^{m \times k}$ for $k \leq n$ and $k \leq m$, respectively.

The *second-order tangent-linear model* (SOTLM) of F is defined as

$$\mathbf{y}^{(1,2)} = F^{(1,2)}(\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(1,2)}) \equiv \underbrace{< \nabla F(\mathbf{x}), \mathbf{x}^{(1,2)} >}_{=0} + < \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} > \tag{1.10}$$

with $\mathbf{x}^{(2)}, \mathbf{x}^{(1)} \in \mathbb{R}^n$, $\mathbf{y}^{(1,2)} \in \mathbb{R}^m$, and $< \nabla F, \mathbf{x}^{(1,2)} >= 0$ for $\mathbf{x}^{(1,2)} = 0$. Thereby,

$$< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} > \equiv << \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >, \mathbf{x}^{(2)} >= \mathbf{y}^{(1,2)}$$

with $\mathbf{y}^{(1,2)}$ represents a *second-order tangent-linear projection* of the Hessian $\nabla^2 F(\mathbf{x})$ defined by Equation (1.12) in directions $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}$, as first-order tangent-linear projection $< A, \mathbf{x}^{(2)} >$ of

$$(a_{k,j})_{j=1,\ldots,n}^{k=1,\ldots,m} \equiv A =< \nabla^2 F, \mathbf{x}^{(1)} > \in \mathbb{R}^{m \times n} \quad \text{with} \quad a_{k,j} =< f''_{k,j,*}, \mathbf{x}^{(1)} >= \sum_{i=1}^{n} f''_{k,j,i} \cdot x_i^{(1)}$$

for $\mathbf{x}^{(1)} \equiv (x_i^{(1)})_{i=1,\ldots,n}$ in direction $\mathbf{x}^{(2)}$. A is the first-order tangent-linear projection of $\nabla^2 F(\mathbf{x})$ in direction $\mathbf{x}^{(1)}$ as defined by Equation (1.4).

The *second-order adjoint model* (SOADM) of F is defined as

$$\mathbf{x}_{(1)}^{(2)} = F_{(1)}^{(2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{y}_{(1)}, \mathbf{y}_{(1)}^{(2)}) \equiv \underbrace{< \nabla F(\mathbf{x})^T, \mathbf{y}_{(1)}^{(2)} >}_{=0} + < \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} > \tag{1.11}$$

with $\mathbf{x}^{(2)}, \mathbf{x}_{(1)}^{(2)} \in \mathbb{R}^n$, and $\mathbf{y}_{(1)}^{(2)}, \mathbf{y}_{(1)} \in \mathbb{R}^m$ and $< \nabla F(\mathbf{x})^T, \mathbf{y}_{(1)}^{(2)} >= 0$ for $\mathbf{y}_{(1)}^{(2)} = 0$. The expression

$$< \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} > \equiv << \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) >, \mathbf{x}^{(2)} >= \mathbf{x}_{(1)}^{(2)}$$

with $\mathbf{x}_{(1)}^{(2)} \equiv (x_{(1)}^{(2)})_{i=1,\ldots,n}$ represents a *second-order adjoint projection* of the Hessian $\nabla^2 F$ in directions $\mathbf{y}_{(1)}, \mathbf{x}^{(2)}$ as first-order tangent-linear projection $< B, \mathbf{x}^{(2)} >$ of

$$(b_{j,i})_{j,i=1,\ldots,n} \equiv B =< \mathbf{y}_{(1)}, \nabla^2 F > \in \mathbb{R}^{n \times n} \quad \text{with} \quad b_{j,i} =< f''_{*,j,i}, \mathbf{y}_{(1)} >= \sum_{k=1}^{m} f''_{k,j,i} \cdot y_{(1)k}$$

for $\mathbf{y}_{(1)} \equiv (y_{(1)j})_{j=1,\ldots,m}$ in direction $\mathbf{x}^{(2)}$. B is the first-order adjoint projection of $\nabla^2 F(\mathbf{x})$ in direction $\mathbf{y}_{(1)}$ defined by Equation (1.6). Thus, the Hessian

$$\nabla^2 F = \nabla^2 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n} \equiv \left(f''_{k,j,i}\right)_{i,j=1,\ldots,n}^{k=1,\ldots,m} \tag{1.12}$$

as the symmetric 3-tensor of the second-order sensitivities $f''_{k,j,i} = \frac{\partial^2 y_k}{\partial x_j \partial x_i}(\mathbf{x})$ of F at point $\mathbf{x}$ can be accumulated using SOTLM and SOADM at the computational cost of $O(n^2) \cdot Cost(F)$ and $O(n \cdot m) \cdot Cost(F)$, respectively. This can be done in the former by letting $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ range over Cartesian basis vectors in $\mathbb{R}^n$. In the latter the same Hessian is accumulated by letting $\mathbf{x}^{(2)}$ and $\mathbf{y}_{(1)}$ range over Cartesian basis vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. In case of $m = 1$, the product of the Hessian matrix $\nabla^2 F \in \mathbb{R}^{n \times n}$ with a vector $\mathbf{x}^{(2)} \in \mathbb{R}^n$ can be performed very cheaply using SOADM of F at the computational cost of $O(1) \cdot Cost(F)$.

1.2 Vertex Elimination on Computational Graphs

The SAC given by Equation (1.2) induces a directed acyclic graph (DAG)

$$G \equiv G(F(\mathbf{x})) = (V, E) \quad \text{with} \quad V = \{1, \ldots, q\} \quad \text{and} \quad E = \{(i, j) : i \prec j\} \quad . \tag{1.13}$$

The vertices are sorted topologically with respect to variable dependence, that is,

$$\forall i,j \in V : (i,j) \in E \Rightarrow i < j \quad .$$

We distinguish between the n independent $X = \{1,\ldots,n\}$ the p intermediate $Z = \{n+1,\ldots,n+p\}$ and the m dependent $Y = \{n+p+1,\ldots,n+p+m\}$ vertices, where $V = X \cup Z \cup Y$.

Assumption 1.2. *The sets X, Z, and Y are mutually disjoint.*

Moreover, we distinguish between live and dead vertices. The former [latter] are those corresponding to alive [dead] SAC variables. Under the assumption that all elemental functions are continuously differentiable in some neighborhood of their arguments all edges (i,j) in DAG can be labeled with the value of the *local partial derivatives*

$$c_{j,i} = c_{j,i}(v_k)_{k \prec j} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \quad . \tag{1.14}$$

This yields the *linearized* computational graph (l-DAG) of F. From now on we use G to refer to the linearized computational graph of F. Thus, the Jacobian ∇F in Equation (1.7) of F can be obtained by the elimination of all intermediate vertices Z of G yielding the bipartite graph

$$\tilde{G} = (\tilde{V}, \tilde{E}) \quad \text{with} \quad \tilde{V} = V - Z \quad \text{and} \quad \tilde{E} = \{(i,j) : i \prec^* j \text{ for } i \in X, j \in Y\} \tag{1.15}$$

as proposed by Griewank and Reese [GR91] based on Baur's interpretation [Bau74] of each Jacobian entry

$$f'_{j-(n+p),i} = \sum_{\pi \in \{i \to j\}} \prod_{(k,l) \in \pi} c_{l,k} \tag{1.16}$$

as the elimination of all paths π connecting an independent vertex $i \in X$ to the dependent vertices $j \in Y$. The correctness of this approach results immediately from the chain rule. One way to achieve this is to eliminate all intermediate vertices [Tad08] from G. Therefore, each successor $k \in S_j$ of an intermediate vertex j is connected to all of its predecessors $i \in P_j$. This corresponds to *back-elimination* of all outedges (j,k) of vertex j, which we denote by

$$G - j \equiv G - (j,k) \quad \forall k \in S_j \quad .$$

For $(i,k) \notin E$ a new edge (i,k) is generated with the label

$$c_{k,i} = c_{k,j} \cdot c_{j,i}$$

as the value of the local partial derivative $\frac{\partial v_k}{\partial v_i}$ of v_k with respect to v_i defined by Equation (1.14). Otherwise the value of $c_{k,i}$ is updated as

$$c_{k,i} += c_{k,j} \cdot c_{j,i} \quad .$$

In the former case *fill-in* is generated whereas *absorption* takes place in the latter. Finally, the vertex j along with all of its incoming and outgoing edges are removed from G as illustrated in Figure 1.1.

Thus, the elimination of all intermediate vertices yields $\tilde{G}$ in Equation (1.15) with labels on remaining edges $\tilde{E}$ representing exactly the nonzero elements of ∇F according to Equation (1.16). In terms of computational complexity we count the number of floating point multiplication (MULS) performed during the elimination procedure that represents an "upper bound for the number of performed additions" that is affected by the elimination ordering as proposed by Naumann [Nau04a]. More precisely, the number

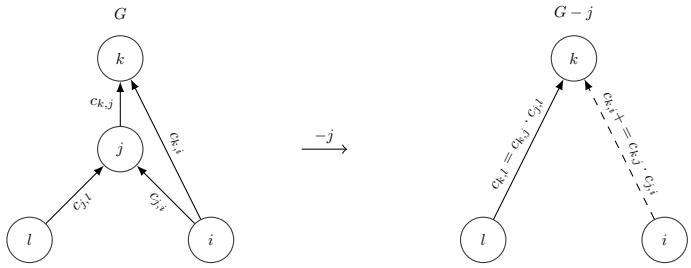


Figure 1.1: Vertex Elimination on DAGs.

of multiplications achieved by the elimination of a single intermediate vertex j is equal to its Markowitz degree [Mar57] as

$$Mark(j) = |P_j| \cdot |S_j| \quad .$$

Thereby, elimination of j is likely to affect the Markowitz degree of its neighbors.

However, Naumann [Nau06] has shown that OPTIMAL JACOBIAN ACCUMULATION (OJA), that is, accumulating the Jacobian of F with minimum number of multiplications is NP-complete. Thereby, he assumes the local partial derivatives attached to the edges of G to be algebraically dependent. However, this is not always the case such that the structural problem remains unsolved.

However, the elimination of an intermediate vertex can be considered as a special case of edge elimination in G as proposed by Naumann [Nau02] with a refined version in form of face elimination proposed by the same author in [Nau04b]. For a comprehensive discussion of the existing elimination techniques we refer the reader to [GW08]. However, the focus of the following will be on Jacobian accumulation using pure vertex elimination technique as the main objective is to tackle the memory bottleneck of the AD in reverse mode as introduced in detail at the beginning of Chapter 2. Thereby, we aim to apply vertex elimination locally to free memory at certain evaluation point of the target function F whenever a given memory bound is reached. Therefore, we consider vertex elimination to be much more suitable.

Given an elimination ordering σ of $p = |Z|$ intermediate vertices the bijective mapping

$$\sigma : \{1, \cdots, p\} \to Z \tag{1.17}$$

denotes a permutation of the elements of Z. Moreover, we denote by

$$G - \sigma = G - [\, \sigma(1), \ldots, \sigma(p) \,]$$

the elimination of vertices Z from G in σ-order. In fact, there are exactly $p!$ different orderings in which the intermediate vertices can be eliminated, where two different elimination orderings very likely yield different number of multiplications and fill-in as well. The latter becomes important when exploiting the sparsity of extended Jacobians as introduced in the following chapter. Two classical orderings are denoted by *forward* and *reverse* that refer to the ascending and decreasing order in which intermediate vertices are visited, respectively. Henceforth, we do eliminate vertices of G in forward and reverse ordering, respectively.

Example 1.1. *In order to support the discussion above let us have a closer look at a very simple example*

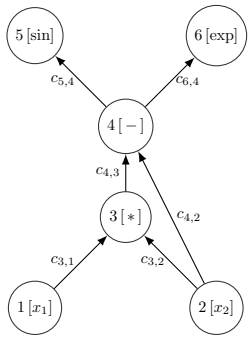
function $F : \mathbb{R}^2 \to \mathbb{R}^2$ given as the following system of non-linear equations.

$$\begin{aligned} \textit{for} \quad & i = 1, \ldots, n \\ & t = x_1 \cdot x_2 - x_2 \\ & x_1 = \sin(t) \\ & x_2 = \exp(t) \end{aligned} \tag{1.18}$$

For $n = 1$ the SAC of F is as follows.

$$\begin{aligned} v_1 &= x_1; \quad v_2 = x_2; \\ v_3 &= v_1 \cdot v_2; \\ v_4 &= v_3 - v_2; \\ t &= v_4; \\ v_5 &= \sin(v_4); \\ v_6 &= \exp(v_4); \\ x_1 &= v_5; \quad x_2 = v_6; \end{aligned}$$

Independent variables x_1 and x_2 are given by the SAC variables v_1 and v_2, respectively. Intermediate SAC variables are given by v_3 and v_4. The latter represents the program variable t, which is used in the SAC statements v_5 and v_6 representing the dependent variables x_1 and x_2, respectively. Hence, v_4 denotes an alive SAC variable. As one can see here, inputs x_1 and x_2 are overwritten at every iteration step i and represent both inputs and outputs. The corresponding l-DAG is shown in Figure 1.2 (a) with edge labels as shown in (b). Consider vertex 3 with two incoming edges $(1, 3)$ and $(2, 3)$ and one outgoing edge $(3, 4)$. The former are labeled with the value of the local partial derivatives $c_{3,1} = x_2$ and $c_{3,2} = x_1$, respectively. The latter is labeled with $c_{4,3} = -1$. Henceforth, whenever we talk about the example function we mean F with the implementation in Example 1.1. Figure 1.3 illustrates the accumulation of



(a)

$$\begin{aligned} c_{3,1} &= v_2; \\ c_{3,2} &= v_1; \\ c_{4,3} &= 1; \\ c_{4,2} &= -1; \\ c_{5,4} &= \cos(v_4); \\ c_{6,4} &= \exp(v_4); \end{aligned}$$

(b)

Figure 1.2: Linearized DAG (a) with the Value of Edge Labels (b).

the Jacobian $\mathbb{R}^{2\times 2} \ni \nabla F(x_1, x_2) = \begin{pmatrix} f'_{1,1} & f'_{1,2} \\ f'_{2,1} & f'_{2,2} \end{pmatrix}$ via forward vertex elimination, where

$$\begin{aligned} f'_{1,1} &= \cos(x_1 \cdot x_2 - x_2) \cdot x_2; \quad f'_{1,2} = \cos(x_1 \cdot x_2 - x_2) \cdot x_1 - \cos(x_1 \cdot x_2 - x_2) \\ f'_{2,1} &= \exp(x_1 \cdot x_2 - x_2) \cdot x_2; \quad f'_{2,2} = \exp(x_1 \cdot x_2 - x_2) \cdot x_1 - \exp(x_1 \cdot x_2 - x_2) \quad . \end{aligned}$$

Thereby, vertex 3 is eliminated by connecting vertices 1 and 2 to vertex 4 resulting in a fill-in $(1,4)$ *and an absorption* $(2,4)$ *labeled as*

$$c_{4,1} = c_{3,1} \cdot c_{4,3} = x_2, \quad \textit{and} \quad c_{4,2} += c_{3,2} \cdot c_{4,3} = -1 + x_1 \quad .$$

Additional elimination of vertex 4 yields the bipartite graph $\tilde{G}$ *resulting in four fill-in* $(1,5)$, $(1,6)$, $(2,5)$, $(2,6)$ *labeled as follows.*

$$c_{5,1} = c_{4,1} \cdot c_{5,4} = x_2 \cdot \cos(x_1 \cdot x_2 - x_2); \quad c_{5,2} = c_{4,2} \cdot c_{5,4} = (-1 + x_1) \cdot \cos(x_1 \cdot x_2 - x_2)$$
$$c_{6,1} = c_{4,1} \cdot c_{6,4} = x_2 \cdot \exp(x_1 \cdot x_2 - x_2); \quad c_{6,2} = c_{4,2} \cdot c_{6,4} = (-1 + x_1) \cdot \exp(x_1 \cdot x_2 - x_2)$$

Analogous, the process of Jacobian accumulation via reverse vertex elimination is illustrated in Figure 1.4. The vertex 4 is eliminated by connecting the vertex 4 to vertices 5 and 6 resulting in fill-in (3,5), (3,6), (2,5), (2,6) labeled as follows.

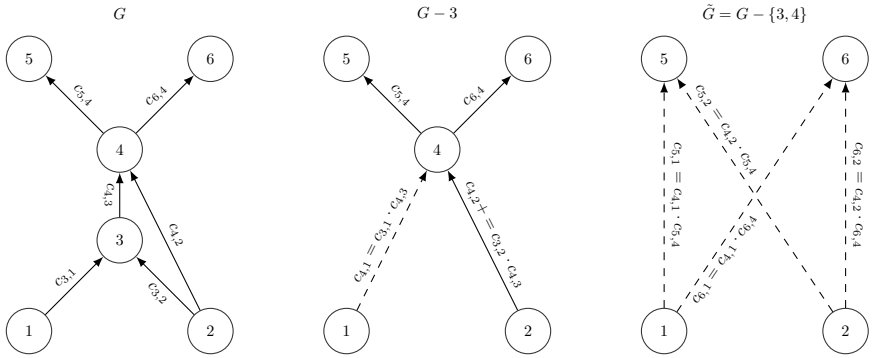


Figure 1.3: Forward Vertex Elimination yielding five Fill-in and one Absorption.

$$c_{5,3} = c_{5,4} \cdot c_{4,3} = \cos(x_1 \cdot x_2 - x_2); \quad c_{6,3} = c_{6,4} \cdot c_{4,3} = \exp(x_1 \cdot x_2 - x_2)$$
$$c_{5,2} = c_{5,4} \cdot c_{4,2} = -\cos(x_1 \cdot x_2 - x_2); \quad c_{6,2} = c_{6,4} \cdot c_{4,2} = -\exp(x_1 \cdot x_2 - x_2)$$

Additional elimination of vertex 3 yields the bipartite graph $\tilde{G}$ *yielding 2 fill-in (1,5), (1,6) and 2 absorptions (2,5), (2,6) labeled as follows.*

$$c_{5,1} = c_{3,1} \cdot c_{5,3} = x_2 \cdot \cos(x_1 \cdot x_2 - x_2)$$
$$c_{5,2} += c_{3,2} \cdot c_{5,3} = -\cos(x_1 \cdot x_2 - x_2) + x_1 \cdot \cos(x_1 \cdot x_2 - x_2)$$
$$c_{6,1} = c_{3,1} \cdot c_{6,3} = x_2 \cdot \exp(x_1 \cdot x_2 - x_2)$$
$$c_{6,2} += c_{3,2} \cdot c_{6,3} = -\exp(x_1 \cdot x_2 - x_2) + x_1 \cdot \exp(x_1 \cdot x_2 - x_2)$$

Finally, the entries $f'_{j,i}$ *with* $i, j \in \{1,2\}$ *of the Jacobian are represented by the labels* $c_{j+4,i}$ *of the edges* $(i, j+4)$ *of the bipartite graphs* $\tilde{G}$ *shown in Figure 1.3 and Figure 1.4, respectively. Hence, the Jacobian of our example function is accumulated with six and eight multiplications in forward and reverse ordering with one and two fill-in, respectively. We note that this does not necessarily mean that forward ordering is generally better that reverse ordering. Indeed, often the opposite is the case in the practice as our experimental results will show.*

The DAG as an intuitive representation of F is widely used in AD to address conceptual as well as runtime and memory [Nau08] issues arising in context of Jacobian accumulation specially by the

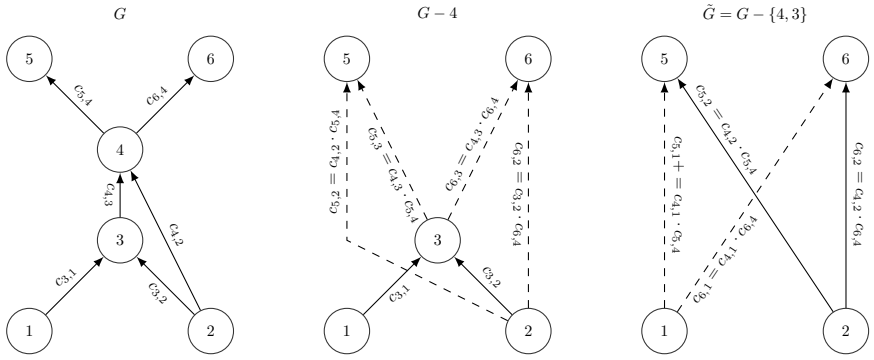


Figure 1.4: Reverse Vertex Elimination yielding six Fill-in and two Absorptions.

application of elimination techniques. Nonetheless, we consider the DAG consisting of vertices and edges as a *dynamic* structure in the way it deals with the memory as explained by Duff et al. [DER86] in the context of sparse linear algebra. For instance, eliminating a vertex means freeing the memory locations of that vertex along with all of its incident edges as well as allocating new memory for fill-in edges as they occur. Thereby, an edge is incident to a vertex, when it has this vertex as target or source vertex. The consequence is that the memory is accessed dynamically by allocation and deallocation instructions during the entire Jacobian accumulation process. In this context, we define by

$$Mem(G) = |E| \cdot \mu_e = \sum_{j=n+1}^{q} |P_j| \cdot \mu_e \tag{1.19}$$

the memory consumption of G in bits with μ_e representing the amount of storage in bits required by an edge. Furthermore, we define edges by their source and target vertices, so that we get

$$\mu_e = 2 \cdot \mu_v + \mu_F \quad ,$$

where μ_v and μ_F denote the amount of storage in bits needed for a vertex and the floating point label of an edge in G, respectively. Thereby, the vertices V of G are implicitly given by the edges as

$$V = \{j : \exists (i,k) \in E \text{ with } j = i \text{ or } j = k\} \quad .$$

However, in the following chapter we introduce a lower triangular matrix referred to as the *extended Jacobian* [TFP03] representation of the SAC of F. The extended Jacobian matrix is supposed to be a static structure. Furthermore, we manage to exploit its sparsity by first detecting the required memory pattern for a given elimination ordering in a symbolic step as explained in Section 2.4.1 and using the resulting memory scheme to accumulate the Jacobian on a statically allocated CRS representation of the extended Jacobian as discussed in Section 2.2. The overhead of detecting memory pattern for a given elimination ordering can be regarded as a preprocessing step when assuming the control flow to be fixed at points of interest in the input domain D. However, in practice the input domain may be decomposed into such intervals, such that any interval changes would require new memory pattern detection. In Section 2.5 we discuss also first ideas on parallelizing the Jacobian accumulation by elimination and present first results on a shared memory architecture. Furthermore, we introduce in Section 2.6 how to deal with the memory bound by keeping track of the memory usage of the underlying data structure and enabling local elimination whenever the given available memory bound is reached.

Chapter 3 is concerned with retrieving the information about constant sensitivities of in general derivative tensors. Therefore, we present runtime algorithms to compute both constants and sparsity pattern of

target tensors. Furthermore, we show how to exploit constants in the process of sparse Jacobian as well as Hessian computations and discuss them on examples. Finally, Section 3.3 presents a fast algorithm to overestimate the Hessian pattern under exploitation of the partial separability of F. The algorithm is then generalized to a recursive one converging to the exact algorithm for sufficiently large recursion level.

Chapter 2

Jacobian Accumulation on Extended Jacobians

2.1 Motivation and Summary of Results

Traditionally, almost any AD software available at the community website `www.autodiff.org` provides forward and reverse modes of AD for calculating the Jacobian matrix $\nabla F(\mathbf{x})$ accurately at a given point $\mathbf{x}$ on some kind of internal representation of F kept on memory or hard disk. In fact, the storage is known to be the bottleneck of AD to handle larger problems in reverse mode. For instance, a tape is the internal representation of choice in the C++ operator overloading tool ADOL-C [GJM+99] that presents an augmented version of F. Thus, ∇F can be obtained in forward and reverse fashion by an interpretative forward and reverse propagation of directional derivatives and adjoints [NMRC07] through the tape, respectively. The forward mode AD can be implemented very cheaply in terms of memory by single forward propagation of directional derivatives at runtime. However, the reverse mode needs to store some data [HNP05] in the so-called forward sweep to allow the data flow reversal [Nau08] needed for backward propagation of adjoints. The latter is recently the focus of ongoing research activities of the AD community for $m = 1$ as a single application of the reverse mode is enough to accumulate the gradient of F efficiently. To handle the memory bottleneck, checkpointing schedules e.g. revolve [GW00] have been developed for time-dependent problems. However, they require user's knowledge in both the function F and the reverse mode AD as well.

To illustrate the memory problem in reverse mode and to demonstrate the idea behind checkpointing let us consider Figure 2.1. Here (a) represents a DAG of the function $f : \mathbb{R}^2 \to \mathbb{R}$

$$\begin{aligned} &\text{for } \; i = 1, \ldots, n \\ &\quad x_1 = \sin(x_1 \cdot x_2 - x_2) \qquad (s_1) \\ &\quad x_2 = \exp(x_1 \cdot x_2 - x_2) \qquad (s_2) \\ &y = x_1 + x_2 \qquad (s_3) \end{aligned}$$

for $n = 2$ as a light modification of our example function of Equation (1.18). Edge labels are missing explicitly just for simplicity. Hence, eliminating intermediate vertices $3, 4, \ldots, 10$ in reverse mode yields the complete bipartite graph as shown in (b) at a cost of twelve multiplications, whereas the forward elimination yields the same gradient but at a total cost of twenty-two multiplications [1]. Now let us assume that there is not enough memory to store the DAG in (a). The reader may agree that given a

[1] The calculation of the respective number of multiplications of both elimination orderings is left to the reader.

memory bound one can find an n, where the resulting DAG would not fit into the memory. Consequently, the reverse mode AD does not seem to be applicable anymore.

A closer look into f figures out that the two statements s_1 and s_2 at each loop iteration $i = 1, \ldots, n$ overwrite the value of the inputs x_1 and x_2, which are their own arguments. We denote the respective values of the inputs after the iteration i using the superscript i. For instance, x_1^2 denotes the value of x_1 after the iteration $i = 2$. Hence, in order to reduce the memory consumption of the reverse mode checkpointing is applied in (c), where $CP(i)$ denotes a checkpoint in iteration i storing the value of the variables x_1 and x_2 before they get overwritten.

For this, the evaluation of f for $n = 2$ at the beginning results in two checkpoints $CP(1)$ and $CP(2)$ along with the DAG consisting only of the vertices $9, 10$ and 11. The latter is the result of *augmented evaluation* of the statement s_3. The augmented evaluation of (a piece of) a function in our context is supposed to generate the respective (piece of) DAG. However, no vertex can be eliminated so far in (c). The total memory consumption is $4 \cdot \mu_F + 2 \cdot \mu_e$ with μ_F and μ_e denoting the amount of storage in bits needed for an edge and a floating point value as explained at the end of the previous chapter. We recapitulate that each of two checkpoints stores only two floating point values.

Now, the augmented evaluation of the for loop for $i = 2$ yields (d) with *locally eliminatable* vertices $7, 8, 9, 10$. A detailed discussion about locally eliminatable vertices is given at the beginning of Section 2.6. However, the correct values of x_1 and x_2 must be read from the checkpoint $CP(2)$ in advance. Hence, $CP(2)$ can be deleted that results in a total memory consumption of $2 \cdot \mu_F + 8 \cdot \mu_e$. We yield (e) by eliminating $10, 9, 8, 7$ at a cost of six multiplications. Analog, we yield (f) by first reading the correct values of x_1 and x_2 from $CP(1)$ followed by the augmented evaluation of s_1 and s_2 for $i = 1$ at a memory cost of $8 \cdot \mu_e$. Finally, (b) results from the elimination of $6, 5, 4, 3$ at a cost of six additional multiplications yielding twelve multiplications in total. This number is identical to the global reverse mode resulting from (a), which is rather random and not the case in general. Thus, ∇f is computed by checkpointing at the lower memory cost of at most $2 \cdot \mu_F + 8 \cdot \mu_e$ instead of $14 \cdot \mu_e$ in (a) for reasonably $\mu_F \leq \mu_e$.

At this point it has to be mentioned that the memory reduction by checkpointing is achieved at the expense of additional (augmented) loop evaluations at each of those checkpoints. In general, the user of checkpointing strategy has to take care of first its applicability to the underlying problem F. This requires the deeper view into the program of F, which in practice can be very large. Revolve, for instance, is designed especially for time-dependent problems with similar structure as our example function f. In this context, the for loop can be regarded as the time iteration, where the checkpointing is applied. However, a major question that matters is about the size of the checkpoints. For our simple example it was easy to figure out which values to store. However, this is in general more than an easy task [HNP05]. Of course conservatively one can store the values of all variables on the left hand side of assignments. With a little imagination can be appreciated that in practice this may also exceed the memory bound. Hence, the conservative checkpointing may turn out to be not feasible in practice at all.

However, "the CHECKPOINTING problem is to determine for a given upper bound on persistent memory K a set of values computed by the single assignment code as defined in Equation (1.2) such that the computational cost of adjoint propagation becomes minimal" as proposed by Naumann in [Nau09]. In this work Naumann shows that DAG REVERSAL problem (DAGR) i.e. finding a reversal scheme that uses at most K memory and $c \leq C$ costs is NP-complete, where C denotes the upper bound on the cost of recomputing some (SAC) values. He shows also the CHECKPOINTING problem to be in fact the same problem as DAGR, which follows the NP-completeness of the former too.

At this point, we hope the reader agrees that the application of checkpointing strategy [SG05, KW06] is not straightforward, despite the fact that not every user wishes to spend the necessary effort. Hence, a black-box tool would be nice even if its performance does not quite reach that of pure reverse mode AD. This is exactly the motivation in the following, where the memory reduction is supposed to be done automatically without any user intervention and expertise in AD. Therefore, vertex elimination is applied

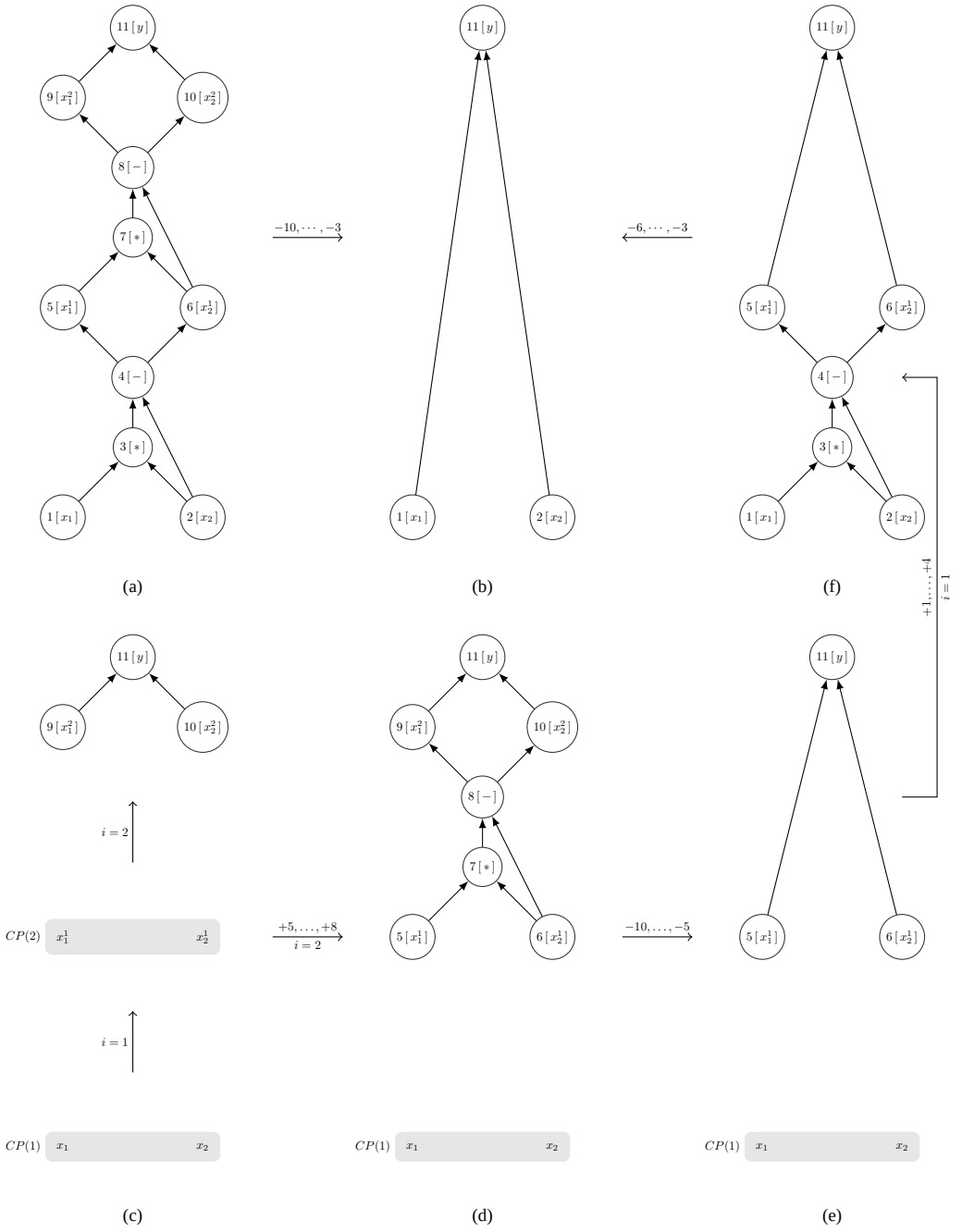


Figure 2.1: Checkpointing Idea on a DAG, where $CP(i)$ for $i = 1, 2$ denotes a checkpoint at loop iteration i storing the value of the variables x_1 and x_2 before getting overwritten. The subscript i to a variable denotes its value after ith iteration. The prefix "-" to a vertex index means that it is eliminated, whereas "+" indicates its Generation.

locally when the given memory bound at certain point of the function (augmented) evaluation is hit. The memory usage is tracked at runtime. Hence, the Jacobian/gradient of F is accumulated while taking care of the memory consumption at runtime. We refer to this approach as *iterative Jacobian accumulation*.

In our case, arbitrary elimination orderings (techniques) such as forward, reverse, or Markowitz-based heuristics [AGN03] can be applied locally. The latter is not considered in this work. As our experimental results will show, the reverse mode exhibits much better runtime results than the forward one for the test cases considered in this work. Thus, we consider our approach using reverse mode AD rather as *local reverse mode AD* during forward evaluation of F.

In the following section we first introduce row elimination on extended Jacobians being conceptually the same as vertex elimination on the respective DAGs. Furthermore, we manage to exploit the sparsity of extended Jacobians using compressed row storage to reduce their quadratic (in number of rows) memory complexity in Section 2.3. The reason for using extended Jacobians as internal representation instead of graphs is to avoid the dynamic memory access affecting the runtime of Jacobian accumulation by vertex elimination. To show this a runtime comparison for computing the gradient $\nabla f \in \mathbb{R}^n$ of the scalar function

$$f : \mathbb{R}^n \to \mathbb{R}, \quad y = f(\mathbf{x}) \quad \text{with} \quad y = \prod_{i=1}^{n} x_i$$

between the reverse mode AD on an early DAG and CRS implementations is presented in Figure 2.2. Thereby, no sparsity is exploited as ∇f is supposed to be a dense vector in $\mathbb{R}^n$. Moreover, we compare the runtime of both CRS and DAG with that of forward finite difference approximation denoted by FFDA. We observe that the elimination on CRS is orders of magnitude faster compared with its DAG counterpart as well as with FFDA. More precisely, for $n = 10000$ the former needs 0.46 seconds to accumulate ∇f instead of 44.25 and 67.76 seconds in case of DAG and FFDA, respectively. Hence, we believe that the runtime loss in case of DAG to be mostly caused by the dynamic memory access. The runtime loss of FFDA against two reverse AD variants (DAG and CRS) lies reasonably in the fact that $n + 1$ function calls are required to accumulate ∇f.

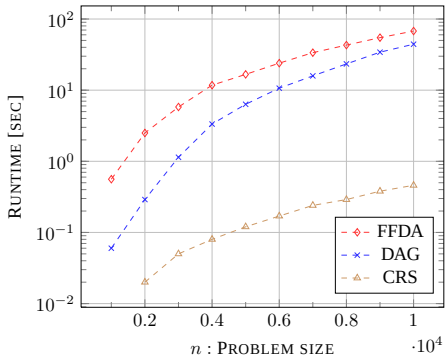


Figure 2.2: Runtimes of Gradient Computation using DAG, CRS, and Finite Differences.

In the following we present first some numerical results in Section 2.4.2 considering the static problem of row elimination on extended Jacobians and their respective CRS. The numerical results show that the sparsity exploitation of extended Jacobians using CRS reduces the memory consumption drastically by a factor of roughly thirty-one. However, we observe at the same time that CRS underperforms compared with the dense extended Jacobian (DEJ) by increase in the problem size because of the linear overhead of searching for dependencies and spots in the former. Henceforth, whenever we talk about a *spot*, we mean a memory unit that is used to store an extended Jacobian entry. At the same time the increase

in problem size has a direct impact on the number of rows of considered matrices meaning even larger search spaces. The impact of the latter becomes more clear when we try to parallelize the process of Jacobian accumulation in Section 2.5. The experimental results in Section 2.5.3 show the most part of the speedup by parallelization to be gained simply by the decomposition resulting in smaller search spaces for dependencies. Hence, by focusing on the static problem, we figure out that the search space has a large impact on the runtime behavior of Jacobian accumulation by elimination on both DEJ and its CRS counterpart, despite the fact that CRS has additionally a linear overhead to search for a particular entry or spot in worst case.

Finally, we introduce in Section 2.6 the iterative approach on DEJ and CRS as well. The numerical results are presented in Section 2.6.3. Thereby, we observe that assignment level elimination exhibits the best memory behavior as shown in Table 2.5 on page 93. However, its runtime on a time-dependent problem turns out to be not really comparable with that of the reverse mode AD implemented in ADOL-C. Nonetheless, we also observe that memory adapting strategy according to Equation (2.24) improves the runtime considerably with negligible loss in memory. The runtime and memory comparisons with ADOL-C are shown in Figure 2.34 (a) and (b), respectively. We note that for this test case DEJ is used instead of its CRS counterpart because of the better runtime performance of the former as shown in Figure 2.34 (c).

2.2 Dense Jacobian Accumulation

The SAC in Equation (1.2) of the function F can be written as a system of nonlinear equations [GW08]

$$C(\mathbf{v}) = (\varphi_j(v_i)_{i \prec j} - v_j)_{j=n+1,\ldots,q} = 0 \tag{2.1}$$

with $\mathbf{v} = (v_1, \cdots, v_q)$ and $q = n + p + m$. Differentiation of Equation (2.1) with respect to $\mathbf{v}$ yields the lower triangular matrix

$$C' = C'(\mathbf{v}) \equiv (c'_{j,i})_{i,j=1,\ldots,q} \quad \text{with} \quad c'_{j,i} = \begin{cases} c_{j,i} & \text{if } i \prec j \\ -1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

referred to as the *extended Jacobian* of F with rows and columns enumerated as $i, j, k \in V$ with V defined by Equation (1.13). Henceforth, for better readability we will replace the -1 on the diagonal entries of C' with the corresponding row indices.

Thereby, row j of C' contains the local partial derivatives $c_{j,i}$ of v_j with respect to all of its arguments v_i with $i \prec j$ as defined in Equation (1.14), where the binary relation $i \prec j$ indicates again the direct dependence of row j on row i on C' if and only if $c_{j,i} \neq 0$. Analog, column j contains the local partial derivatives $c_{k,j}$ of all v_k with respect to v_i, which have v_i as their arguments with $j \prec k$. The extended Jacobian and the DAG of F correspond to each other in such a way that a row/column j of C' corresponds to the DAG vertex j. Moreover, a local partial derivative $c_{j,i}$ [$c_{k,j}$] represents the label of the incoming [outgoing] edge (i, j) [(j, k)] to [from] vertex j. In the following we refer to a row/column j as independent for $j \in X$, as intermediate for $j \in Z$, and as dependent for $j \in Y$. For simplicity henceforth we only talk about row elimination. Thus, analog to G, the Jacobian can also be accumulated by row elimination on C'. Therefore, the elimination of a particular intermediate vertex on G is interpreted as the elimination of the corresponding row on C'. We eliminate an intermediate row j by eliminating all nonzero entries $c_{k,j}$ with $j \prec k$. Thereby, $c_{k,j}$ is eliminated by performing Equation (2.2) for all nonzero $c_{j,i}$ of row j while generating fill-in and fill-out for $c_{k,i} = 0$ and $c_{k,i} \neq 0$, respectively. We refer to this a *back-elimination* of the entry $c_{k,j}$ on C'. A gain the terminologies forward and reverse are used to refer to the ascending and decreasing order of intermediate rows, respectively. Thus, the elimination of

an intermediate vertex j by back-elimination of its out-edges on G is interpreted as the elimination of row j via back-elimination of all nonzeros $c_{*,j}$ on the column j. Hence, forward [reverse] vertex elimination on G corresponds to the forward [reverse] row elimination on C'.

The following summarizes all transformations needed for row elimination on C'.

Definition 2.1. *The elimination of row j on the extended Jacobian C' of F with $j \in Z$, $i \in \{1, \ldots, j-1\}$, and $k \in \{j+1, \ldots, q\}$ denoted by $C' - j$ is defined as follows.*

$$c_{k,i} = c_{k,i} + c_{k,j} \cdot c_{j,i} \quad \forall k : j \prec k \quad \textit{and} \quad \forall i : i \prec j \tag{2.2}$$

$$c_{k,j} = 0 \quad \forall k : j \prec k \tag{2.3}$$

$$c_{j,i} = 0 \quad \forall i : i \prec j \tag{2.4}$$

Note that partial derivatives of v_k with respect to v_i during the elimination of row j are computed according to the chain rule in 2.2. Hence, any sensitivities of v_k on any of the v_j with $j \prec k$ as well as of any of the v_j on v_i with $i \prec j$ are removed in Equation (2.3) and Equation (2.4), respectively. **fill-out** are generated. Moreover, for $c_{k,i} = 0$ then 2.2 lead to **fill-in** otherwise they yield **absorption**.

In the following, we introduce Algorithm 2.1 for Jacobian accumulation by row elimination on extended Jacobians that we refer to as *dense Jacobian accumulation by row elimination* (DJARE). It describe the general process of Jacobian accumulation by row elimination on the extended Jacobian of F for a given elimination ordering $\sigma(1), \ldots, \sigma(|Z|)$ with σ as defined by Equation (1.17). At this point, it should be made clear that the algorithms introduced below can be considered as special cases of Gaussian elimination known in context of sparse linear algebra [DER86].

As described in Algorithm 2.2 an intermediate row $j = \sigma(i)$ with $i \in \{1, \cdots, p\}$ is eliminated via back-elimination of all nonzero entries $c'_{k,j} \neq 0$ with $j < k$ according to Equations 2.2 and 2.3. Hence, all nonzeros of row j are set to zero in lines 3-7 of Algorithm 2.1 after back-eliminating all $c'_{k,j}$ according to Equation (2.4). We note that, we use the notation $c'_{\cdot,\cdot}$ explicitly to denote the entries of the extended Jacobian in our algorithms.

Thus, the elimination of all intermediate rows in σ order yields the *eliminated extended Jacobian*

$$\tilde{C'} = C' - \sigma \equiv C' - [\, \sigma(1), \ldots, \sigma(p) \,] \tag{2.5}$$

containing exactly the entries $f'_{j-(n+p),i} = c'_{j,i}$ for $i \in X$, $j \in Y$ of the Jacobian ∇F, which can be extracted by Algorithm 2.4. Obviously, forward [reverse] row elimination can be considered as special cases of Algorithm 2.1 with σ representing the ascending [decreasing] ordering of Z.

Considering the elimination of an intermediate row j in forward ordering the **search space** in line 1 of Algorithm 2.2 can be restricted to the dependent rows Y, since all rows $k \in Z$ with $j \prec k$ are eliminated before j and hence $c'_{k,j} = 0$. The termination of the process of Jacobian accumulation by row elimination introduced above is stated by Lemma 2.1.

Lemma 2.1. *Given an elimination ordering σ of a finite set Z of intermediate rows the process of Jacobian accumulation by row elimination described in Algorithm 2.2 terminates.*

Proof. The proof of termination of Algorithm 2.2 follows immediately from the termination of the corresponding vertex elimination process as a special case of edge elimination on the DAG of F as shown by Naumann [Nau99, Nau04a]. □

Algorithm 2.1 (`JRowElim` (C', σ) : Jacobian by Row Elimination)**.**

Require: : extended Jacobian C' and the elimination ordering σ of Z.
Ensure: : C' after elimination of all intermediate rows in σ order.

1: **for** $j = \sigma(1)$ **to** $\sigma(p)$ **do**
2: `RowElim` $(\mathtt{C}', \mathtt{j})$
3: **for** $i = 1$ **to** $j - 1$ **do**
4: **if** $c'_{j,i} \neq 0$ **then**
5: $c'_{j,i} = 0$
6: **end if**
7: **end for**
8: **end for**

Algorithm 2.2 (`RowElim` $(\mathtt{C}', \mathtt{j})$: Row Elimination)**.**

Require: : extended Jacobian C' and the row index $j \in Z$.
Ensure: : C' after elimination of the intermediate row j.

1: **for** $k = q$ **to** $j + 1$ **do**
2: **if** $c'_{k,j} \neq 0$ **then**
3: `BackElim` $(\mathtt{C}', \mathtt{k}, \mathtt{j})$
4: $c'_{k,j} = 0$
5: **end if**
6: **end for**

Algorithm 2.3 (`BackElim` $(\mathtt{C}', \mathtt{k}, \mathtt{j})$: Back-Elimination)**.**

Require: : extended Jacobian C' and the indices $j, k \in V$ with $j \prec k$.
Ensure: : C' after back-elimination of $c'_{k,j}$.

1: **for** $i = 1, .., j - 1$ **do**
2: **if** $c'_{j,i} \neq 0$ **then**
3: **if** $c'_{k,i} \neq 0$ **then**
4: $c'_{k,i} += c'_{k,j} \cdot c'_{j,i}$
5: **else**
6: $c'_{k,i} = c'_{k,j} \cdot c'_{j,i}$
7: **end if**
8: **end if**
9: **end for**

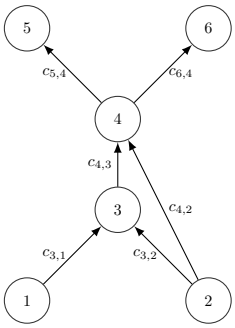
Algorithm 2.4 (`JExtract` $(\mathtt{C}', \nabla\mathtt{F})$: Jacobian Extraction)**.**

Require: extended Jacobian C' of F.
Ensure: the values of the Jacobian ∇F.

1: **for** $j = 1$ **to** m **do**
2: **for** $i = 1$ **to** n **do**
3: $f'_{j,i} = c'_{j+n+p,i}$
4: **end for**
5: **end for**

Example 2.1. *At this point let us have a look again at our example function with G and C' as shown in Figure 2.3 (a) and (b), respectively. In the following explanations we focus on the extended Jacobian.*

Nonetheless, the corresponding DAG transformations are also presented to clarify the relation between these two structures. Independent and dependent rows are $1, 2$ *and* $5, 6$*, respectively. Intermediate rows are given by* 3 *and* 4*. The latter represents the program variable* t*. Both rows* 5 *and* 6 *depend on row* 4 *as* $c_{5,4} \neq 0$ *and* $c_{6,4} \neq 0$*. Considering the corresponding linearized DAG the local partial derivatives* $c_{5,4}$ *and* $c_{6,4}$ *are the labels of the outgoing edges* $(4, 5)$ *and* $(4, 6)$ *from vertex* 4*, respectively.* $c_{4,3}$ *is the label of the incoming edge* $(3, 4)$ *to vertex* 4*. The elimination of vertex* 3 *via single back-elimination of the entry* $c_{4,3}$ *is demonstrated in Figure 2.4 (b). Thereby, fill-in and absorption are generated as* $c_{4,1} = c_{4,3} \cdot c_{3,1}$ *and* $c_{4,2} += c_{4,3} \cdot c_{3,2}$*, respectively. A fill-out is generated as* $c_{4,3} = 0$*.*

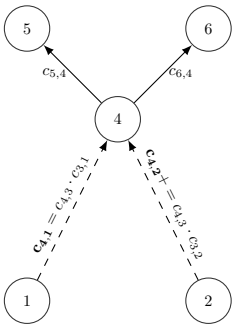


(a)

$$\begin{array}{cccccc} 1 & & & & & \\ 0 & 2 & & & & \\ c_{3,1} & c_{3,2} & 3 & & & \\ 0 & c_{4,2} & c_{4,3} & 4 & & \\ 0 & 0 & 0 & c_{5,4} & 5 & \\ 0 & 0 & 0 & c_{6,4} & 0 & 6 \end{array}$$

(b)

Figure 2.3: Linearized DAG (a) corresponding to the extended Jacobian (b).



(a)

$$\begin{array}{cccccc} 1 & & & & & \\ 0 & 2 & & & & \\ \mathbf{0} & \mathbf{0} & 3 & & & \\ \mathbf{c_{4,1}} & \mathbf{c_{4,2}} & \mathbf{0} & 4 & & \\ 0 & 0 & 0 & c_{5,4} & 5 & \\ 0 & 0 & 0 & c_{6,4} & 0 & 6 \end{array}$$

(b)

Figure 2.4: Elimination of Vertex and Row 3 on G (a) and C' (b), respectively.

Figure 2.5 (a) and (b) illustrate the forward and reverse row elimination, respectively. In (a), row 3 is eliminated as first by single back-elimination of $c_{4,3}$*, resulting in a fill-in* $c_{4,1}$ *and an absorption* $c_{4,2}$ *as*

$$c_{4,1} = c_{4,3} \cdot c_{3,1}, \quad \textit{and} \quad c_{4,2} += c_{4,3} \cdot c_{3,2} \quad .$$

Fill-out are generated as $c_{4,3} = 0, c_{3,1} = 0$*, and* $c_{3,2} = 0$*. Additional elimination of row 4 yields the eliminated extended Jacobian* $\tilde{C}' = C' - [3, 4]$*, finalizing the process of Jacobian accumulation by forward row elimination. Likewise, elimination of row 4 (b) by back-elimination of the entries* $c_{5,4}$ *and*

$c_{6,4}$ yields four fill-in as

$$c_{5,3} = c_{5,4} \cdot c_{4,3}, \quad c_{6,3} = c_{6,4} \cdot c_{4,3} \quad c_{5,2} = c_{5,4} \cdot c_{4,2}, \quad \textit{and} \quad c_{6,2} = c_{6,4} \cdot c_{4,2} \quad .$$

Furthermore, fill-out are generated as $c_{5,4} = 0, c_{6,4} = 0, c_{4,2} = 0$, and $c_{4,3} = 0$. Additional elimination of row 3 yields the eliminated extended Jacobian $\tilde{C}' = C' - [4,3]$ that results in four fill-in as

$$c_{5,1} = c_{5,4} \cdot c_{4,1}, \quad c_{5,2} = c_{5,4} \cdot c_{4,2}, \quad c_{6,1} = c_{6,4} \cdot c_{4,1}, \quad \textit{and} \quad c_{6,2} = c_{6,4} \cdot c_{4,2} \quad .$$

This finalizes the process of Jacobian accumulation by reverse row elimination. Hence, accumulating the Jacobian of our example function in forward and reverse elimination ordering results in totally six and eight multiplications, respectively. The former results in five fill-in one absorption and seven fill-out, whereas the latter yields eight fill-in and eight fill-out. Hence, the entries $f'_{j,i}$ with $i, j \in \{1, 2\}$ of the Jacobian are represented by the entries $c_{j+4,i}$ in the corresponding eliminated extended Jacobian $\tilde{C}'$.

As discussed at the end of the previous chapter, the linearized DAG as a data structure to accumulate Jacobians by vertex elimination, is dynamic in terms of memory access. In opposite, the extended Jacobian as a sub-diagonal matrix is considered as a *static data structure*, as fill-in and fill-out do not cause any memory allocation and deallocation during the elimination step, respectively. However, the main problem using extended Jacobians as the internal representation is their quadratic memory usage, since memory is also allocated for all those *fixed zeros* remaining zero over the entire elimination process. However, we aim to exploit the sparsity of extended Jacobians using the CRS representation to reduce the memory consumption [VNL06], which is discussed in very detail in Section 2.4. In this context, fill-in results in additional memory allocation, whereas fill-out represents memory getting freed during the elimination process. At this point it must be made clear that the focus of the following is not on finding an elimination ordering that minimizes the fill-in. Our goal is rather on finding approaches to reusing fill-out for fill-in for a given elimination sequence.

Herley [Her93] proposes in a unpublished manuscript that finding a vertex elimination ordering minimizing the number of fill-ins on DAGs in the context of Jacobian accumulation is NP-complete. His work bases on early works on minimizing fill-in [Yan81] in the Gaussian elimination process in sparse linear systems formulated by Rose and Tarjan [RT78] as a vertex elimination problem on directed graphs. A note on the NP-completeness of this problem is given by Gilbert [Gil80]. Hence, it seems to be very unlikely to find an elimination ordering minimizing the fill-in in a polynomial time.

2.3 Trading Fill-Out for Fill-In

As already mentioned, the process of Jacobian accumulation by row elimination on extended Jacobians can result in

- **fill-in** by changing zero entries to nonzeros,
- **fill-out** by changing nonzero entries to zeros, and
- **absorptions** by updating nonzero entries.

Consider the extended Jacobian of our example function shown at the top of Figure 2.5. Forward and reverse row elimination result in five and six fill-ins, respectively. This results in five and six additional memory spots in the corresponding CRS representations, which will be illustrated in Section 2.4 in more detail. At the same time the former resp. latter yields seven resp. eight fill-out spots, which can potentially be reused to store fill-in entries as we will discuss below. Thus, the main focus in the following is on reusing fill-out for fill-in as much as possible to reduce the memory consumption of the CRS representation of extended Jacobians. Henceforth, whenever we talk about a *spot*, we mean a memory unit that is used to store an extended Jacobian entry.

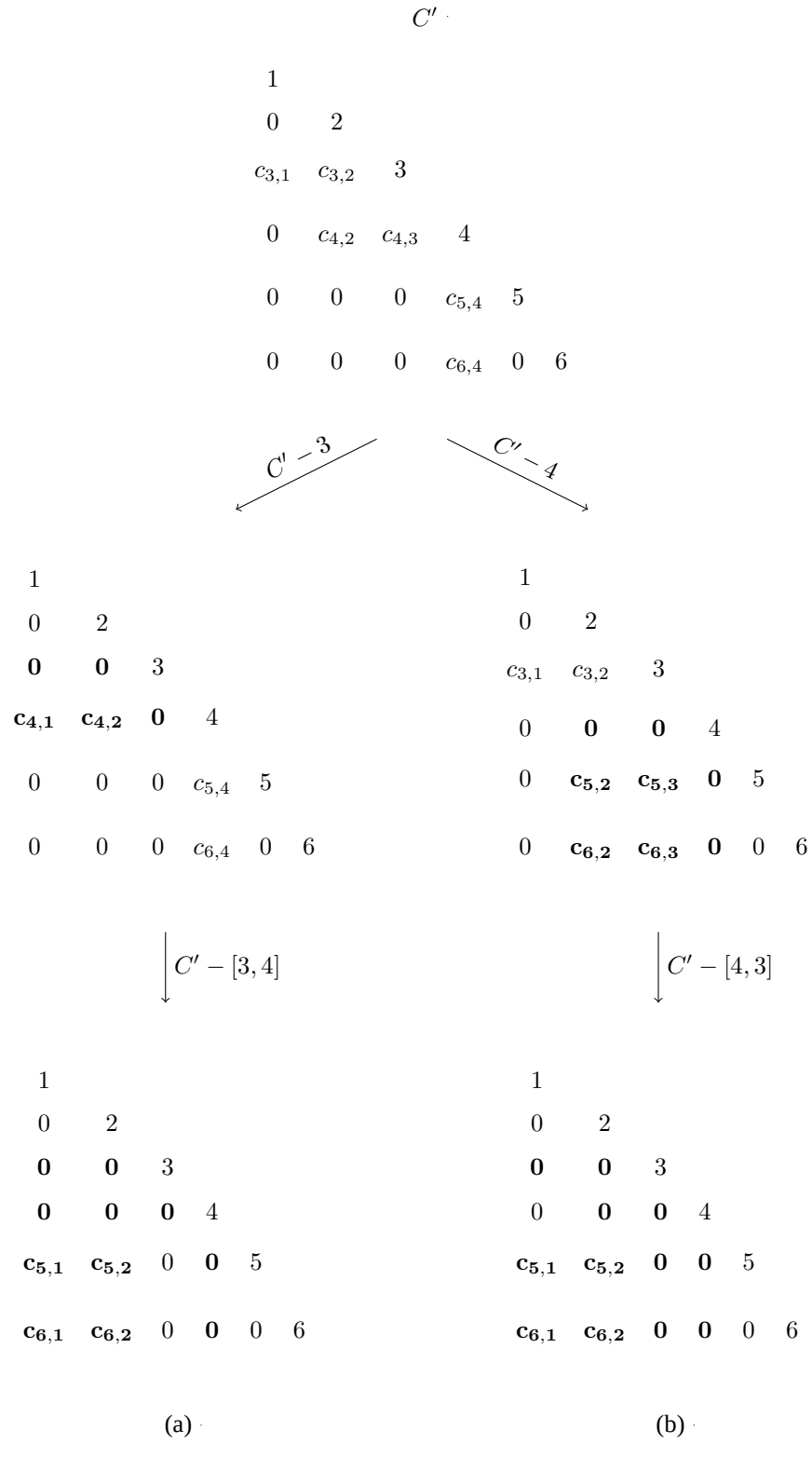


Figure 2.5: Forward (a) and Reverse (b) Row Elimination on Extended Jacobian, respectively.

Notation Summary 2.1. *We use the following symbols to classify the extended Jacobian entries with respect to the type of their spots that we need to analyze the memory pattern resulting from the application of elimination orderings in combination with fill-out exploitation techniques.*

- $*$ *identifies an initial nonzero element.*
- $+$ *denotes the absorption of an initial nonzero element.*
- $\circledast$ *represents a fill-in.*
- $\oplus$ *marks the absorption of a fill-in.*
- $\otimes$ *marks a fill-out reused for fill-in.*
- $\circledcirc$ *marks a fill-out.*

We introduce in the following two ideas for reusing fill-out. Here, we assume that we want to eliminate the intermediate row i.

- **Technique 1** exploits the fact that the elimination of any $c_{j,i}$ yields fill-out in the current memory location. Consider the situation shown in Figure 2.6 (a), where the memory spot of $c_{j,i}$ can be reused to store the fill-in $c_{j,k} = c_{j,i} \cdot c_{i,k}$ as shown in (b), where $c_{j,l}$ is absorbed as $c_{j,l} += c_{j,i} \cdot c_{i,l}$.

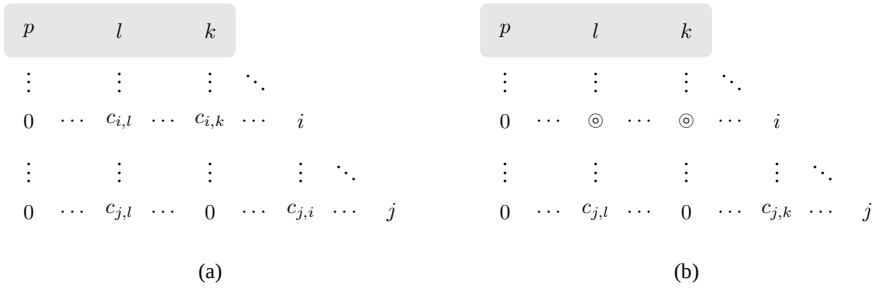


Figure 2.6: Memory Pattern before (a) and after (b) the Application of Technique 1.

- **Technique 2** exploits the existence of sub-diagonal nonzero entries $c_{j,i} \neq 0$ with $j = i + 1$ representing the dependency $i \prec j$ between two neighboring rows i and j also referred to as *immediate successors* as shown in Figure 2.7 (a). After the elimination of $c_{j,i}$ the entire row i becomes zero. Hence, fill-out in row i can be reused to store fill-in generated in row j as shown in Figure 2.7 (b). As a consequence row j expands into row i. We denote this by setting the diagonal entry of the row i to j. Obviously, the absorption $c_{j,k} += c_{j,i} \cdot c_{i,k}$ could also be placed in the spot for $c_{i,k}$ to avoid **memory fragmentation** in order to achieve better cache efficiency [Tad08].

  It is worth mentioning that the elimination of row i via back-elimination of entries $c_{k,i} \neq 0$ with $i \prec k$ needs to assure the correctness of the calculated partial derivatives as the spot of $c_{i,l}$ is reused for fill-in $c_{j,l} = c_{i,l} \cdot c_{j,i}$. Nevertheless, this can be either achieved by saving $c_{i,l}$ before it gets overwritten or eliminating $c_{j,i}$ as the last dependency on i. We note that the initial (ascending) ordering of row entries may get destroyed by the application of Technique 2, which implies a **linear search after the dependencies** over the column index space. An example situation is given when eliminating row i in (b), where $c_{j,l}$ stored in spot of $c_{i,l}$ appears before $c_{j,h}$ with $h < l$. However, this can be avoided by rearranging nonzeros as

$$c_{j,h} \to c_{i,l}, \qquad c_{j,l} \to c_{i,k}, \qquad c_{j,k} \to c_{i,n}, \qquad \text{and} \qquad c_{j,n} \to c_{j,i} \quad .$$

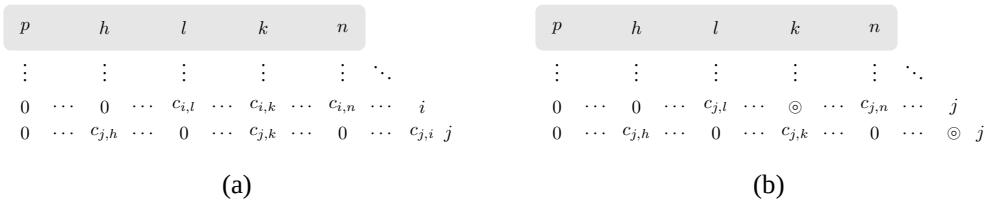


Figure 2.7: Memory Pattern before (a) and after (b) the Application of Technique 2.

Figure 2.8 illustrates the application of fill-out reusing Techniques 1 and 2 introduced above during forward (a) and reverse (b) row (b) elimination. The elimination of row 3 in the former reuses fill-out according to the Technique 2. Furthermore, the application of Techniques 1 and 2 during the elimination of row 4 in addition results in totally one fill-in and three fill-out. Analog, the elimination of row 4 in reverse fashion reuses fill-out according to the Techniques 1 and 2. Additional application of the Technique 2 during the elimination of row 3 yields totally two fill-in and four fill-out. Hence, the total number of necessary spots for accumulating the Jacobian of our example function by row elimination in forward and reverse orderings is seven and eight, respectively. We remember that eleven and twelve fill-in are generated without fill-out exploitation as shown in Figure 2.5 (a) and (b), respectively, in forward and reverse modes. Obviously, the most memory savings are achieved by the application of Technique 2, which is considered in the following in more detail.

Maximum Immediate Successor Enumeration Problem

In order to maximize the number of reused fill-out spots according to Technique 2, one has to find an ordering of the rows of C' that maximizes the number of immediate successors. We formulate this as the MAXIMUM IMMEDIATE SUCCESSOR ENUMERATION (MISE) problem on the corresponding DAG of F. Therefore, we consider a topological ordering

$$top : V \to \{1, \ldots, |V|\} \tag{2.6}$$

of the DAG vertices V, where $top(i) = k$ is the topological index of vertex $i \in V$. Our main goal is to find a topological ordering referred to as *MISE-ordering* of V such that the number of the *immediate successor edges* (i, j) with $top(j) = top(i) + 1$ is maximal. Theorem 2.1 states the NP-completeness of the MISE problem. The proof idea was inspired by Andrew Lyons[2].

Theorem 2.1. *Given a directed acyclic graph $G = (V, E)$ with integer vertices V. The maximum immediate successor enumeration of graph vertices V is NP-complete.*

Proof. Let $G^* = (V, E^*)$ be the transitive closure of G. It is well-known that there is a bidirectional mapping between G^* and the corresponding partially ordered set [Sta00] $(V, <_p)$ also referred to as *poset* on the vertices V of G^*. Let

$$r : V \to \{1, \cdots, |V|\}$$

denote a linear extension of vertices V, such that

$$\forall\, i, j \in V : i <_r j \Leftrightarrow r(i) < r(j) \quad ,$$

[2] http://www.mcs.anl.gov/lyonsam

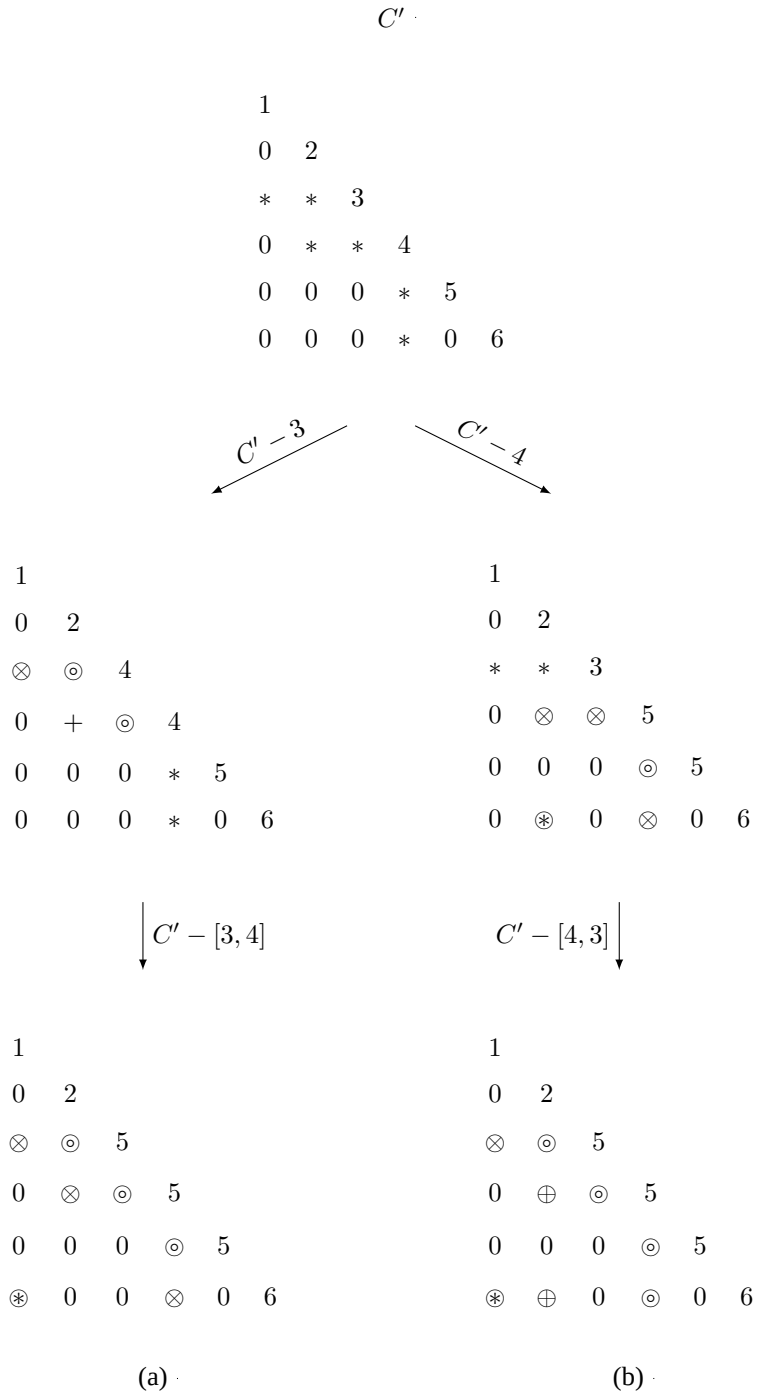


Figure 2.8: Reusing Fill-out in Forward (a) Reverse (b) Row Elimination. The symbols $\odot, \otimes, \oplus$, and $\circledast$ denote fill-out, reused fill-out, absorption of a fill-in, and fill-in, respectively.

where r preserves the topological ordering of graph vertices such that

$$i <_p j \Rightarrow i <_r j \quad .$$

Considering now two consecutive vertices i and j in the linear expansion with $r(j) = r(i) + 1$, we have the following two cases:

- $(i, j) \in E^*$ is a step (an immediate successor edge), or
- $(i, j) \notin E^*$ is a jump, otherwise.

The jump [step] number $\sigma(r)$ [$\omega(r)$] of G is the minimum [maximum] number of jumps [steps] in the linear extension r. Chein and Habib explained in [CH80], that any two DAGs with the same transitive closure are equivalent with respect to the jump number problem. It follows then that it can be considered as a problem on the corresponding transitive closure G^* and poset $(V, <_p)$, respectively. Note that every consecutive pair of vertices in r must either be a jump or a step. In particular, we have

$$|\sigma(r)| + |\omega(r)| = |V| - 1 \quad .$$

Thus, any linear extension that maximizes the number of steps, that is, the number of immediate successor edges will also minimize the number of jumps. Hence, the JUMP NUMBER PROBLEM is obviously the MISE problem. Pulleyblank [Pul82] has shown that determining the jump number of a poset is NP-complete. Hence, it follows also that the MISE problem is NP-complete. □

Thus, it seems to be unlikely to find an exact MISE-ordering in polynomial time. However, in the following we introduce a modified version (TopSortAll) of the exponential recursive algorithm proposed by Knuth [KS74] that finds all topological arrangement of vertices of a DAG. In addition to that, we discuss a first idea on reducing the runtime complexity using a branch and bound [Tal06] algorithm (TopSortBB) as follows. We note that proof of concept implementation of the ideas below is beyond the scope of this work.

- **TopSortAll** computes all topological arrangements and picks out the one with the maximum number of immediate successors at the end. The algorithm is exponential in the number of inputs in worst case.
- **TopSortBB** uses the branch and bound idea to reduce the complexity of TopSortAll by finding a criterion to cut the recursion at the level that doesn't lead to a MISE-Ordering of vertices. Therefore, let a^l denote the number of the immediate successors among the visited vertices at the recursion level $l \in \{1, \cdots, |V|\}$. Furthermore, $b^l = |V| - l$ denotes the maximum possible number of the immediate successor edges among the remaining vertices. The idea is to cut the recursion at the level l, if $a^l + b^l \leq c$. Thereby, c denotes the maximum number of immediate successors in G that is supposed to be initially zero.

Consider G and C' of our example function as shown in Figure 2.3 (a) and (b), respectively. The possible topological orderings of graph vertices [extended Jacobian rows/columns] are the following.

$$\begin{aligned} top_1 &: \ 1, 2, 3, 4, 5, 6 \\ top_2 &: \ 1, 2, 3, 4, 6, 5 \\ top_3 &: \ 2, 1, 3, 4, 5, 6 \\ top_4 &: \ 2, 1, 3, 4, 6, 5 \end{aligned}$$

Obviously, G consists of totally two maximum immediate successor edges namely $(3, 4)$ and $(4, 5)$ or $(4, 6)$ over all four possible orderings above from which top_1 represents the initial one of the DAG by creation. One can easily figure out that all four topological orderings are equivalent in terms of fill-out exploitation. Hence, there is no need of reordering G or C' for this example. The resulting fill-out exploitation schemes of the initial ordering has already been shown in Figure 2.8.

2.4 Sparse Jacobian Accumulation

As discussed at the beginning of this chapter rows of the extended Jacobian correspond to SAC variables defined in Equation (1.2). Hence, a row can initially consist of nonzero elements (local partial derivatives) in the number of the parameters of the respective elemental function. For instance, consider the extended Jacobian of example function shown in Figure 2.3 with the SAC given in Example 1.1. Thereby, the row 3 results from the multiplication of two variables, whereas the unary operation $\sin$ yields the row 5.

In the following we illustrate the process of Jacobian accumulation by row elimination under exploitation of the sparsity of extended Jacobians [VNL06], which we refer to as *sparse Jacobian accumulation by row elimination* (SJARE). It consists of two main steps, namely *symbolic* and *accumulation*. The former is concerned with memory prediction by simulating the elimination process on an integer representation of the *sparsity pattern* of C' defined by Equation (2.9) at the point of interest $\mathbf{x}$. After termination of the symbolic elimination process a corresponding static Compressed Row Storage (CRS) [DER86] is allocated. The symbolic step is the focus of Section 2.4.1. The accumulation step uses the CRS allocated by the former to accumulate the Jacobian at $\mathbf{x}$ by initiating the elimination process. But, this time the elimination happens on real data, that is, on the initialized CRS with the values of "initial" local partial derivatives at $\mathbf{x}$ by evaluating F at $\mathbf{x}$. The result of the accumulation step is hence the Jacobian of F at $\mathbf{x}$.

We note that the extended Jacobian along with its CRS representation are runtime-dependent in the sense that they depend on the control flow of F that often is assumed to be fixed, which holds for a bunch of real world numerical applications. It is worth mentioning that actually this fact is the main motivation for more and less any sparse approach. However, in general, changes in inputs may change the control flow of F and hence "potentially" change the sparsity pattern of the underlying extended Jacobian. The latter would imply that a new symbolic step has to be performed to get a valid [GJM$^+$99] memory pattern at the respective point. But, changes in control flow do not necessarily have to lead to changes in the sparsity pattern of C'. It is, to some extent, possible that the latter remains unchanged, while the former changes. This is even more likely when just focusing on the number of spots of rows, regardless of the orders. Here, changes in the sparsity pattern of rows can be tolerated as long as their total number of spots does not change. To clarify this let us consider the example situation given in Figure 2.9. For simplicity let assume that only j depends only on i. Let us consider $C'(F(\mathbf{x}_1))$ as the extended Jacobian resulting by evaluating F at $\mathbf{x}_1$. The elimination of the row i results in $C'(F(\mathbf{x}_1)) - i$ with one additional spot for fill-in $c_{j,k}$. Hence, the row j requires three spots in total. Now, let assume that the evaluation of F at another point $\mathbf{x}_2$ results in $C'(F(\mathbf{x}_2))$ with a different dependency pattern compared to $C'(F(\mathbf{x}_1))$. However, eliminating i here yields the same amount of spots in total as on $C'(F(\mathbf{x}_1))$.

More importantly, the memory pattern resulting from the symbolic step is only valid for the given elimination ordering. Different elimination orderings may require different amounts of memory. Two classical orderings, forward and reverse, have been illustrated in Figure 2.5 with totally eleven (a) and twelve (b) memory spots, respectively. The former [latter] results in CRS representation that is used in Example 2.2 [2.3] to accumulate the target Jacobian.

In the following we use CRS consisting of

- a floating point value vector α,
- an integer column index vector κ, and
- an integer row position vector ρ

to exploit the sparsity of extended Jacobians. The value vector α contains row-wise the nonzero entries of C' with the corresponding column indices stored in κ. Hence, both α and κ vectors are of the same length that we denote by nz, which in our case represents the total number of memory spots detected by the symbolic step. Thereby, the column index of the element $\alpha(i)$ is stored in $\kappa(i)$ for $i = 1, \cdots, nz$. The vector ρ is of length $q+1$ with $\rho(q+1) = nz+1$ marking the end of the last extended Jacobian row. $\rho(i)$

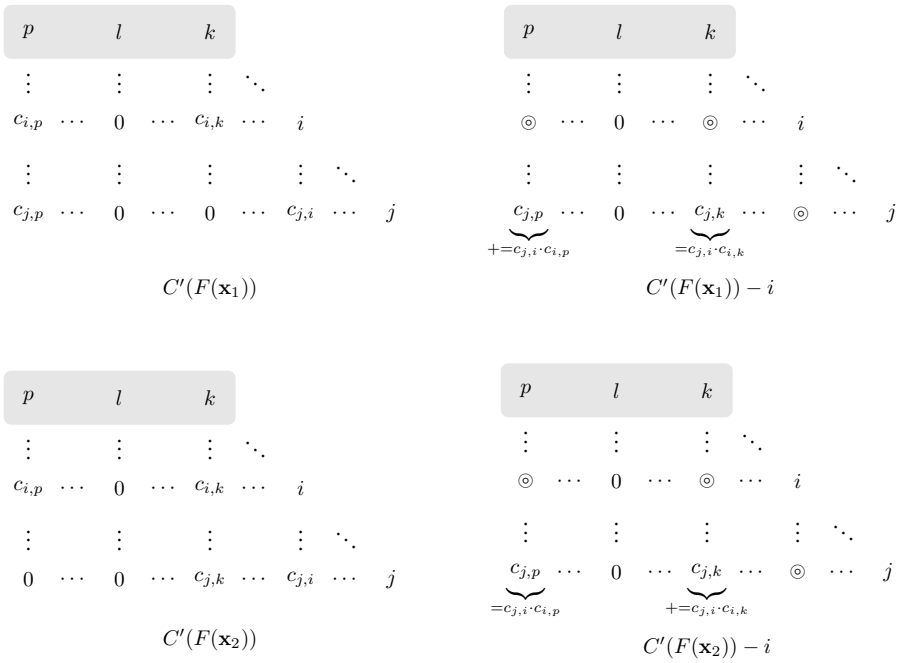


Figure 2.9: Identically Amount of Storage on two structurally different Extended Jacobians.

with $i \in \{n+1, \cdots, q\}$ contains the position of the first nonzero element of row i. The first n elements of ρ correspond to the independent rows, which are initialized to zero. Obviously, the length of a row $i \in V$ can be gained by $\rho(i+1) - \rho(i)$. Henceforth, we denote the CRS representation by (α, κ, ρ). At this point let us have a look at the initial extended Jacobian of our example function as shown in Figure 2.3 (b). Its CRS representation is as follows.

$$\begin{aligned} \alpha &= (c_{3,1}, c_{3,2}, c_{4,2}, c_{4,3}, c_{5,4}, c_{6,4}) \\ \kappa &= (1, 2, 2, 3, 4, 4) \\ \rho &= (0, 0, 1, 3, 5, 6, 7) \end{aligned}$$

Rows 1 and 2 are independent, hence $\rho(1) = \rho(2) = 0$. The first nonzero entry of row 3 is stored in $\alpha(1)$, hence $\rho(3) = 1$. Similarly, $\rho(4) = 3$ points to $\alpha(3)$ containing the first nonzero element of the row 4 with the column index stored in $\kappa(3)$. The difference $\rho(4) - \rho(3)$ yields two as the length of row 3. However, as discussed previously fill-in has to be taken into account to provide enough memory needed for Jacobian accumulation on CRS representation of extended Jacobians. Analog to Equation (2.5) for extended Jacobians the elimination of all intermediate rows in the given order σ yields the *eliminated CRS*

$$(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho}) = (\alpha, \kappa, \rho) - \sigma \equiv (\alpha, \kappa, \rho) - [\, \sigma(1), \ldots, \sigma(p) \,] \; .$$

Example 2.2. *The following CRS is used to accumulate the Jacobian ∇F of our example function by forward row elimination. Fill-in spots $\alpha(3)$ of row 4, $\alpha(6), \alpha(7)$ of row 5, and $\alpha(9), \alpha(10)$ of row 6 are*

initialized to zero at the beginning of the elimination process. Hence, we get the following initial CRS.

$$\begin{aligned}\alpha &= (c_{3,1}, c_{3,2}, 0, c_{4,2}, c_{4,3}, 0, 0, c_{5,4}, 0, 0, c_{6,4})\\ \kappa &= (1, 2, 1, 2, 3, 1, 2, 4, 1, 2, 4)\\ \rho &= (0, 0, 1, 3, 6, 9, 12)\end{aligned}$$

- *Elimination of row* 3 *yields* $(\alpha, \kappa, \rho) - 3$ *as*

$$\begin{aligned}\alpha &= (0, 0, \mathbf{c_{4,1}}, \mathbf{c_{4,2}}, 0, 0, 0, c_{5,4}, 0, 0, c_{6,4})\\ \kappa &= (0, 0, 1, 2, 3, 1, 2, 4, 1, 2, 4)\\ \rho &= (0, 0, 1, 3, 6, 9, 12)\end{aligned}$$

  with $c_{4,1} = c_{3,1} \cdot c_{4,3}$ *and* $c_{4,2} += c_{3,2} \cdot c_{4,3}$.

- *Elimination of row* 4 *yields* $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho}) = (\alpha, \kappa, \rho) - [3, 4]$ *as*

$$\begin{aligned}\alpha &= (0, 0, 0, 0, 0, \mathbf{c_{5,1}}, \mathbf{c_{5,2}}, 0, \mathbf{c_{6,1}}, \mathbf{c_{6,2}}, 0)\\ \kappa &= (0, 0, 0, 0, 0, 1, 2, 0, 1, 2, 0)\\ \rho &= (0, 0, 1, 3, 6, 9, 12)\end{aligned}$$

  with $c_{5,1} = c_{4,1} \cdot c_{5,4}, c_{5,2} = c_{4,2} \cdot c_{5,4}, c_{6,1} = c_{4,1} \cdot c_{6,4}$, *and* $c_{6,2} = c_{4,2} \cdot c_{6,4}$.

Example 2.2 illustrates the accumulation of the Jacobian of the example function by row elimination in forward ordering as described in Algorithm 2.5.

Algorithm 2.5 (`JRowElim` $((\alpha, \kappa, \rho), \sigma)$: Jacobian by Row Elimination)**.**

Require: CRS representation (α, κ, ρ) of C' and the elimination ordering σ.
Ensure: (α, κ, ρ) after elimination of all intermediate rows in σ order.

1: **for** $j = \sigma(1)$ **to** $\sigma(p)$ **do**
2: `RowElim` $((\alpha, \kappa, \rho), \mathtt{j})$
3: **for** $l = \rho(j)$ **to** $\rho(j+1) - 1$ **do**
4: **if** $\alpha(l) \neq 0$ **then**
5: $\alpha(l) = 0$
6: $\kappa(l) = 0$
7: **end if**
8: **end for**
9: **end for**

Therefore, row $j = 3$ in line 2 is eliminated by back-elimination of the entry $\alpha(5) = c_{4,3}$ as shown in line 4 of Algorithm 2.6.

Algorithm 2.6 (`RowElim` $((\alpha, \kappa, \rho), \mathtt{j})$: Row Elimination)**.**

Require: : CRS representation (α, κ, ρ) of C' and the row index $j \in Z$.
Ensure: : (α, κ, ρ) after elimination of the intermediate row j.

1: **for** $k = q$ **to** $j - 1$ **do**
2: $l =$ `Find` $((\alpha, \kappa, \rho), \mathtt{k}, \mathtt{j})$
3: **if** $l > 0$ **and** $\alpha(l) \neq 0$ **then**

```
4:    BackElim((α, κ, ρ), k, j, l)
5:    α(l) = 0
6:    κ(l) = 0
7:   end if
8:  end for
```

As shown, for instance, in line 2 of Algorithm 2.6 a naive linear index search as described in Algorithm 2.8 is used to find the dependency $c_{k,j}$. Thereby, row k depends on row j if

$$l \geq 0 \quad \text{and} \quad \kappa(l) == j \quad \text{and} \quad \alpha(l) \neq 0 \quad .$$

Algorithm 2.7 (`BackElim` $((\alpha, \kappa, \rho), \mathtt{k}, \mathtt{j}, \mathtt{l})$: Back-Elimination)**.**

Require: : CRS representation (α, κ, ρ) of C' and $j, k \in V$ with $j \prec k$ and $l \in \{\rho(k), \ldots, \rho(k) - 1\}$.
Ensure: : (α, κ, ρ) after elimination of $c_{k,j}$.

```
1: for l_1 = ρ(j) to ρ(j+1) − 1 do
2:  if l_1 > 0 and α(l_1) ≠ 0 then
3:   l_2 = Find((α, κ, ρ), k, κ(l_1))
4:   if l_2 > 0 then
5:    if κ(l_2) == κ(l_1) then
6:     α(l_2) += α(l_1) · α(l)
7:    else
8:     α(l_2) = α(l_1) · α(l)
9:     κ(l_2) = κ(l_1)
10:   end if
11:  end if
12: end if
13: end for
```

Algorithm 2.8 (`Find` $((\alpha, \kappa, \rho), \mathtt{j}, \mathtt{i})$)**.**

Require: (α, κ, ρ) of C' and the indices $j, i \in V$ with $i < j$.
Ensure: Position $l \geq 1$ if exists l such that $\kappa(l) == i$, or $\kappa(l) == 0$, otherwise $l = 0$ of the element $c_{j,i}$ in CRS.

```
1: p = 0
2: found = false
3: for l = ρ(j) to ρ(j+1) − 1 do
4:  if κ(l) == 0 and found == false then
5:   p = l
6:   found = true
7:  end if
8:  if κ(l) == i then
9:   return l
10:  end if
11: end for
12: if p == 0 then
13:  print ERROR : CRS Invalidity!
14: end if
```

15: **return** p

The algorithm returns an integer value larger than zero as the position of the target entry in α, otherwise zero meaning that no spot is allocated for the target entry. However, this indicates that CRS is not valid at the current point for the reasons have been discussed at the beginning of this section. As an alternative, binary index search can also be applied when the ascending ordering of κ during the entire elimination process is guaranteed. This is for example the case when not reusing fill-out for fill-in according to the Technique 2 as discussed in Section 2.3. As only row $k = 4$ depends on row 3, single back-elimination of $\alpha(5) = c_{4,3}$ is enough to eliminate row 3 yielding the fill-in $\alpha(3) = \mathbf{c_{4,1}} = c_{3,1} \cdot c_{4,3}$ with $\kappa(3) = 1$, the absorption $\alpha(4) = \mathbf{c_{4,2}} += c_{3,2} \cdot c_{4,3}$ with $\kappa(4) = 2$, and the fill-outs $\alpha(5) = \kappa(5) = 0, \alpha(1) = \kappa(1) = 0$, and $\alpha(2) = \kappa(1) = 0$. Additional elimination of row 4 yields $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho})$ containing the nonzero entries of ∇F as

$$f'_{1,1} = \alpha(6) = c_{5,1}, \quad f'_{1,2} = \alpha(7) = c_{5,2}, \quad f'_{2,1} = \alpha(9) = c_{6,1}, \quad \text{and} \quad f'_{2,2} = \alpha(10) = c_{6,2}\ .$$

Algorithm 2.9 (`JExtract` $((\alpha, \kappa, \rho), \nabla\mathtt{F})$: Jacobian Extraction)**.**

Require: (α, κ, ρ) and the zero Jacobian $\nabla F = 0$.
Ensure: the Jacobian ∇F with numerical values.

1: **for** $j = 1$ **to** m **do**
2: **for** $i = 1$ **to** n **do**
3: $l = \mathtt{Find}\,((\alpha, \kappa, \rho), \mathtt{j} + \mathtt{n} + \mathtt{p}, \mathtt{i})$
4: **if** $l > 0$ **and** $\kappa(l) == i$ **then**
5: $f'_{j,i} = \alpha(l)$
6: **end if**
7: **end for**
8: **end for**

Thus, the accumulation of the example Jacobian in forward ordering needs totally eleven spots instead of fifteen that are needed to store the entire sub-diagonal matrix C'. Thus, we save four memory spots for this little example. However, our experimental results show that the savings are more substantial for larger problems. Once the elimination process terminates Algorithm 2.9 can be used to extract the Jacobian ∇F from $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho})$. Analog, Example 2.3 illustrates the reverse row elimination with a memory consumption of totally twelve spots as shown in Figure 2.5 (b).

Example 2.3. *The following CRS is used to accumulate ∇F of our example function by reverse row elimination. Fill-in spots $\alpha(5), \alpha(6), \alpha(7)$ of the row 5, and $\alpha(9), \alpha(10), \alpha(11)$ of the row 6 are initialized to zero.*

$$\begin{aligned}\alpha &= (c_{3,1}, c_{3,2}, c_{4,2}, c_{4,3}, 0, 0, 0, c_{5,4}, 0, 0, 0, c_{6,4})\\ \kappa &= (1, 2, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4)\\ \rho &= (0, 0, 1, 3, 5, 9, 13)\end{aligned}$$

- *Elimination of column 4 yields $(\alpha, \kappa, \rho) - 4$ as*

$$\begin{aligned}\alpha =&(c_{3,1}, c_{3,2}, 0, 0, 0, \mathbf{c_{5,2}}, \mathbf{c_{5,3}}, 0, 0, \mathbf{c_{6,2}}, \mathbf{c_{6,3}}, 0)\\ \kappa =&\ (1, 2, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4)\\ \rho =&(0, 0, 1, 3, 5, 9, 13)\end{aligned}$$

with $c_{5,2} = c_{4,2} \cdot c_{5,4}, c_{5,3} = c_{4,3} \cdot c_{5,4}, c_{6,2} = c_{4,2} \cdot c_{6,4}$, and $c_{6,3} = c_{4,3} \cdot c_{6,4}$.

- *Elimination of column* 3 *yields* $(\alpha, \kappa, \rho) - [4, 3]$ *as*

$$\begin{aligned}\alpha &= (0, 0, 0, 0, \mathbf{c_{5,1}}, \mathbf{c_{5,2}}, 0, 0, \mathbf{c_{6,1}}, c_{6,2}, 0, 0)\\ \kappa &= (1, 2, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4)\\ \rho &= (0, 0, 1, 3, 5, 9, 13)\end{aligned}$$

with $c_{5,1} = c_{3,1} \cdot c_{5,3}, c_{5,2} += c_{3,2} \cdot c_{5,3}, c_{6,1} = c_{3,1} \cdot c_{6,3}$, *and* $c_{6,2} += c_{3,2} \cdot c_{6,3}$.

Definition 2.2. *Given an extended Jacobian* C' *and a CRS representation* (α, κ, ρ)*. The memory consumption of* C' *and* (α, κ, ρ) *are defined as*

$$Mem(C') = \sum_{i=1}^{q} (i-1) \cdot \mu_F \tag{2.7}$$

and

$$Mem(CRS) = Mem(\alpha) + Mem(\kappa) + Mem(\rho)\ , \tag{2.8}$$

respectively. Thereby, $Mem(\alpha) = nz \cdot \mu_F$*,* $Mem(\kappa) = nz \cdot \mu_I$*, and* $Mem(\rho) = (q+1) \cdot \mu_I$*, where* μ_F *and* μ_I *denote the number of bits for floating-point and integer data types, respectively.* $Mem(\cdot)$ *is assumed to return the memory size of the argument data type in bits. Obviously, the smaller the number of entire nonzeros (*nz*) the bigger the memory savings for CRS compared to its dense representation. In opposite, memory saving shrinks with increasing* nz *that in worst case may end up with* $Mem(CRS) > Mem(C')$ *because of memory overhead of* κ *and* ρ*. However, such situation are unlikely at least for the test cases considered here as discussed in Section 2.4.2.*

So far we have discussed the Jacobian accumulation process on compressed row storage representation of extended Jacobians under the assumption that the given memory pattern is valid at the point of interest. However, the focus of the following section is on symbolic step. Therefore, we proposes algorithms to predict the memory requirement for CRS representation of extended Jacobians for a given elimination ordering. We note again that conservatively any variation in inputs that changes the sparsity pattern of C' requires new memory detection. However, the memory usage remains unchanged at all those points, where the sparsity pattern of C' does not change. Obviously, one and the same CRS can be used to accumulate the Jacobian at all those points.

2.4.1 Symbolic Elimination

In the following we present conceptual algorithms and discuss them with the help of examples that are used in this work to predict the memory pattern required for Jacobian accumulation on CRS of extended Jacobians, where the resulting memory scheme depends very much on the given elimination ordering σ. Therefore, we use the *bit pattern*

$$BP = BP(C')$$

representation of the extended Jacobian C', which can be obtained from its *sparsity pattern*

$$P = P(C') \equiv (p_{j,i})_{i,j=1,\ldots,q} \quad \text{with} \quad p_{j,i} \in \{0, 1\} \tag{2.9}$$

with 1's denoting nonzero entries. Each row $j \in V$ of BP corresponds to row j of P. The latter is decomposed into $b_j = \left\lceil \frac{j}{\mu_I} \right\rceil$ blocks of length μ_I as the number of integer bits. $BP(j, k)$ with $k \in \{1, \cdots b_j\}$ stores the integer value represented by block k. The direct dependence of row j on row i on BP is given as

$$i \prec j \Leftrightarrow p_{j,i} = 1 \Leftrightarrow BP(j, b_i)\,\&\,2^e = 1\ ,$$

where $b_i = \left\lceil \frac{i}{\mu_I} \right\rceil$ and $e = (i-1) \ \% \ \mu_I$ with & resp. % denoting bit-wise AND resp. OR operators as explained in Notation Summary 2.2. Henceforth, we consider P and BP as equivalent and prefer to use $i \prec j$ to denote the dependency of row j on row i on BP whenever appreciate. Moreover, every row j consists of one additional element as $BP(j, k+1)$ to store its total number of required memory spots.

Definition 2.3. *The memory consumption $Mem(BP)$ of the bit pattern BP is defined as*

$$Mem(BP) = \sum_{j=1}^{q} (b_j + 1) \cdot \mu_I \ , \tag{2.10}$$

where $b_j = \left\lceil \frac{j}{\mu_I} \right\rceil$ denotes the number of μ_I-blocks of row $j \in V$.

Notation Summary 2.2. *The following symbols are used in the context of symbolic elimination algorithms.*

$|$ *represents bit-wise OR operation.*

$||$ *represents logical OR operation.*

$\%$ *represents the modulus operation.*

$\&$ *represents bit-wise AND operation.*

$\lceil \cdot \rceil$ *represents round up operation.*

$\&\&$ *represents logical AND operation.*

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ 1 & 1 & 3 & & & \\ 0 & 1 & 1 & 4 & & \\ 0 & 0 & 0 & 1 & 5 & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{6} \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \\ 3 & 2 & \\ 6 & 2 & \\ 8 & 0 & 1 \\ \mathbf{8} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

(a) (b)

Figure 2.10: Sparsity Pattern P (a) and the corresponding 4-bit Integer Bit Pattern BP (b).

As an example let us consider the sparsity pattern and its 4-bit integer [3] i.e. $\mu_I = 4$ bit pattern representation of the extended Jacobian of our example function shown in Figure 2.10. Thereby, we have

$$\begin{aligned} &BP(3,1) = 2^0 + 2^1 = 3; \quad BP(3,2) = 2; \\ &BP(4,1) = 2^1 + 2^2 = 6; \quad BP(4,2) = 2; \\ &BP(5,1) = 2^3 = 8; \quad BP(5,2) = 0; \quad BP(5,3) = 1; \\ &\mathbf{BP(6,1) = 2^3 = 8}; \quad \mathbf{BP(6,2) = 0}; \quad \mathbf{BP(6,3) = 1} \ . \end{aligned}$$

[3] We consider 4-bit integers just for illustration purposes. Realistic number are 32-bit and 64-bits integers depending on the underlying hardware.

Consider row 6 of BP, it is decomposed into $\lceil \frac{6}{4} \rceil = 2$ blocks of 4-bit integers. Hence, the first block yields the integer value $8 = 2^3$ stored in $BP(6,1)$, whereas the second one is 0. $BP(6,3) = 1$ indicating that row 6 requires initially a single memory spot. The dependency $4 \prec 6$ is given as $BP(6,1)$ & $2^3 = 1$. Thus, the detection of the memory consumption of SJARE in a given order σ can be performed by the application of symbolic row elimination to the initial bit pattern BP yielding eliminated bit pattern

$$\tilde{BP} = BP - \sigma := BP - [\,\sigma(1), \ldots, \sigma(p)\,]\,.$$

Here, the initialization of the bit pattern is performed during the evaluation process of the underlying function using Algorithm 2.15 at runtime. Moreover, Algorithm 2.10 eliminates rows of BP in σ-order by symbolic back-elimination of their nonzero entries. Henceforth, we use the notation *symbolic forward* and *symbolic reverse* to refer to the symbolic forward and symbolic reverse row elimination on the bit pattern, respectively. The former and latter detect the required amount of memory for accumulating the Jacobian on the respective CRS in forward and reverse ordering as illustrated in Example 2.2 and Example 2.3, respectively. The respective memory detections for the former and latter are illustrated in Example 2.4 and Example 2.5, respectively.

We note that the symbolic algorithm needs to take the memory spots of fill-out into account. One way to do this is by keeping the corresponding 1's that we refer to as *fill-out 1's* in BP to yield the entire memory usage of the given elimination ordering at the end of the symbolic elimination as described by Algorithm 2.10. This enables us to keep nonzero entries of rows in CRS in ascending order, which is the case here. Moreover, this would allow more efficient binary index search over kappa entries than the linear one presented in Algorithm 2.8 under, however the assumption that the ordering remains unchanged over entire elimination process. However, we will consider in the following the latter as it is also used in context of iterative approach to deal with the memory bound. In that context keeping fill-out 1's is not necessary as the ordering of the kappa elements is not required as explained in much more detail in Section 2.6. In particular, it is enough to maintain the maximum number of nonzeros of rows over entire iterations. However, how much improvement on SJARE the binary search would contribute remains an open question.

Now, as an example let us consider the elimination of row 3 on P shown in Example 2.4. Bold 1's such as $p_{4,1}$ represent fill-in. Fill-out 1's corresponding to $p_{3,1}$, $p_{3,2}$, and $p_{4,3}$ remain unchanged. Thus, they have to be ignored in further elimination process. For instance, additional elimination of row 4 should avoid the generation of fill-in $p_{5,3}$ and $p_{6,3}$, since $p_{4,3}$ represents a fill-out 1. In other words, fill-out 1's should not be interpreted as dependencies during the elimination. Doing this, we get only four fill-ins (instead of six) by the elimination of row 4 that results in total memory spots of eleven. The identification of a fill-out bit in BP that corresponds to a 1 in the respective sparsity pattern P can be done by introducing a Boolean vector

$$D \in \{false, true\}^q$$

of length q used inAlgorithm 2.10 to mark eliminated rows. D is assumed to be initially false. After the elimination of row $j \in \{\sigma(1) \ldots, \sigma(p)\}$, we mark row j as eliminated by $D(j) = true$ as shown in line 3. Hence, a $p_{j,i} = 1 \in P$ with $j \in \{n+1, \ldots, q\}$, and $i \in \{1, \ldots, j-1\}$ represents a fill-out if and only if $D(j) = true$ or $D(i) = true$. The proof follows immediately from Equation (2.3) and Equation (2.4), where the elimination of a row j results in fill-outs $c_{k,j} = 0$ and $c_{j,i} = 0$ for all $j \prec k$ and $i \prec j$, respectively. With other words, after the elimination of the row j all nonzeros of row and column j are set to zero; hence they denote fill-out.

We note that the implementation of symbolic algorithms introduced here might be different. In particular, we duplicate bit pattern rows to avoid element-wise fill-in detection as shown in line 5 of Algorithm 2.12. Therefore, the first instance of a row is supposed to keep the real dependencies, whereas the other contains fill-out 1's additionally that would enable a much faster block-wise binary OR (|) over bit

pattern rows. However, this doubles the memory consumption of bit pattern as our experimental results will show.

Example 2.4. *We illustrate in the following symbolic forward row elimination as described in Algorithm 2.10 on BP shown in Figure 2.10 (b) that yields the memory pattern of the CRS used in Example 2.2.*

1. *Elimination of row 3 yields $BP - 3$ with $\mathbf{7} = \mathbf{2^0} + 2^1 + 2^2$ as follows.*

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ 1 & 1 & 3 & & & \\ \mathbf{1} & 1 & \mathbf{1} & 4 & & \\ 0 & 0 & 0 & 1 & 5 & \\ 0 & 0 & 0 & 1 & 0 & 6 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \\ 3 & 2 & \\ \mathbf{7} & \mathbf{3} & \\ 8 & 0 & 1 \\ 8 & 0 & 1 \end{pmatrix}$$

$$P - 3 \qquad\qquad BP - 3$$

2. *Elimination of row 4 yields $BP - [3,4]$ with $\mathbf{11} = \mathbf{2^0} + \mathbf{2^1} + 2^3$ as follows.*

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ 1 & 1 & 3 & & & \\ \mathbf{1} & 1 & 1 & 4 & & \\ \mathbf{1} & \mathbf{1} & 0 & 1 & 5 & \\ \mathbf{1} & \mathbf{1} & 0 & 1 & 0 & 6 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \\ 3 & 2 & \\ 7 & 3 & \\ \mathbf{11} & 0 & \mathbf{3} \\ \mathbf{11} & 0 & \mathbf{3} \end{pmatrix}$$

$$P - [3,4] \qquad\qquad BP - [3,4]$$

Hence, forward row elimination on the CRS of our example function requires $11 = 2 + 3 + 3 + 3$ spots.

Algorithm 2.10 (`JSRowElim` $(\mathtt{BP}, \mathtt{D}, \sigma)$: Memory Prediction for SJARE)**.**

Require: bit pattern BP, initially false Boolean vector D of length q, and the elimination ordering σ.
Ensure: BP after the symbolic elimination of all intermediate rows in σ order.

1: **for** $j = \sigma(1)$ **to** $\sigma(p)$ **do**
2: `SRowElim` $(\mathtt{BP}, \mathtt{D}, \mathtt{j})$
3: $\mathtt{D(j)} = \mathtt{true}$
4: **end for**

Algorithm 2.11 (`SRowElim`$(\mathtt{BP}, \mathtt{D}, \mathtt{j})$: Symbolic Row Elimination)**.**

Require: bit pattern BP of the extended Jacobian C'.
Ensure: BP after symbolic elimination of row j.

1: $e = (j-1) \,\%\, \mu_I$
2: $b_j = \left\lceil \frac{j}{\mu_I} \right\rceil$
3: **for** $k = q$ **to** $j-1$ **do**
4: **if** $BP(k, b_j) \& 2^e == 1$ **and** $D(k) == false$ **then**
5: $\texttt{SBackElim}\,(\texttt{BP}, \texttt{D}, \texttt{j}, \texttt{k}, \texttt{b}_\texttt{j})$
6: **end if**
7: **end for**

Algorithm 2.12 ($\texttt{SBackElim}(\texttt{BP}, \texttt{D}, \texttt{j}, \texttt{k}, \texttt{b}_\texttt{j})$: Symbolic Back Elimination)**.**

Require: BP, row index j, block index l, and bit position m.
Ensure: filled bit pattern BP after front-elimination of the dependency $((l-1)\cdot\mu_I + m) \prec j$ with

1: **for** $l = 1$ **to** b_j **do**
2: **for** $m = 0$ **to** $\mu_I - 1$ **do**
3: $i = (l-1)\cdot\mu_I + m$
4: **if** $D(i) == false$ **and** $BP(j,l)\&2^m == 1$ **and** $BP(k,l)\&2^m == 0$ **then**
5: $BP(k,l) = BP(k,l) \mid 2^m$
6: $BP(k, \left\lceil \frac{k}{\mu_I} \right\rceil + 1) = BP(k, \left\lceil \frac{k}{\mu_I} \right\rceil + 1) + 1$
7: **end if**
8: **end for**
9: **end for**

Example 2.5. *We illustrate in the following the symbolic reverse row elimination according to Algorithm 2.10 on BP shown in Figure 2.10 (b) yielding the memory pattern of the CRS used in Example 2.3.*

1. *Elimination of column 4 yields $BP - 4$ with $\mathbf{14} = \mathbf{2^1} + \mathbf{2^2} + 2^3$ as follows.*

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ 1 & 1 & 3 & & & \\ 0 & 1 & 1 & 4 & & \\ 0 & \mathbf{1} & \mathbf{1} & 1 & 5 & \\ 0 & \mathbf{1} & \mathbf{1} & 1 & 0 & 6 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \\ 3 & 2 & \\ 6 & 2 & \\ \mathbf{14} & 0 & 3 \\ \mathbf{14} & 0 & 3 \end{pmatrix}$$

$P - 4$ $\qquad$ $BP - 4$

2. *Symbolic elimination of column 3 yields $BP - [4,3]$ with $\mathbf{15} = \mathbf{2^0} + 2^1 + 2^2 + 2^3$ as follows.*

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ 1 & 1 & 3 & & & \\ 0 & 1 & 1 & 4 & & \\ \mathbf{1} & 1 & 1 & 1 & 5 & \\ \mathbf{1} & 1 & 1 & 1 & 0 & 6 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \\ 3 & 2 & \\ 6 & 2 & \\ \mathbf{15} & 0 & \mathbf{4} \\ \mathbf{15} & 0 & \mathbf{4} \end{pmatrix}$$

$P - [4,3]$ $\qquad$ $BP - [4,3]$

Hence, reverse column elimination on CRS of our example function requires $12 = 2 + 2 + 4 + 4$ *spots.*

Algorithm 2.13 describes the CRS construction after the termination of the symbolic elimination procedure `SJRowElim`$(\texttt{BP}, \texttt{D}, \sigma)$ described in Algorithm 2.10. The lines 3, 11, and 12 call the routine `allocate`$(\texttt{v}, \texttt{len})$, which allocates the memory for vector $v = \alpha, \kappa, \rho$ of the length len. Since no local partial derivatives are evaluated in symbolic mode α is initialized to zero as shown in line 21. On the contrary, both κ and ρ vectors as shown in lines 5, 10, 15, 22, and 28 are initialized properly according to the memory pattern given by $\tilde{BP}$. Thereby, $\rho(q+1) = len + 1$ in line 10 marks the end of qth row. For a nonzero row j, $\rho(j)$ in line 15 is initialized to the current counter c. The counter incrementation of line 23 yields $\rho(j+1) = c + nz_j$ denoting the start position of the next row $j+1$, where nz_j represents the number of nonzeros of row j. For $nz_j = 0$ the counter remains unchanged and thus we set $\rho(j) = c$ as shown in line 28. Is worth mentioning that the corresponding κ part of each nonzero row j initialized in line 22 is in ascending order. Furthermore, we save the initial ordering of κ elements in κ_{save}, which is used to reuse CRS for the accumulation ∇F at another point of interest assuming the CRS validity in terms of memory pattern for that point. Once CRS is constructed Algorithm 2.14 can be used to insert local partial derivatives into CRS.

Algorithm 2.13 (`ConstructCRS`$(\texttt{BP}, (\alpha, \kappa, \rho), \kappa_{\texttt{save}})$: CRS Construction)**.**

Require: Bit pattern BP containing the amount of spots for CRS (α, κ, ρ).
Ensure: Initialized (α, κ, ρ) and κ_{save}.

1: $c = 1$
2: $len = 0$
3: `allocate`$(\rho, \texttt{q}+\texttt{1})$
4: **for** $i = 1$ **to** n **do**
5: $\quad \rho(i) = 0;$
6: **end for**
7: **for** $j = n+1$ **to** q **do**
8: $\quad len = len + BP(j, \left\lceil \frac{j}{\mu_I} \right\rceil + 1)$
9: **end for**
10: $\rho(q+1) = len + 1$
11: `allocate` $(\alpha, \texttt{len})$
12: `allocate` $(\kappa, \texttt{len})$
13: **for** $j = n+1$ **to** q **do**
14: $\quad$ **if** $BP(j, \left\lceil \frac{j}{\mu_I} \right\rceil + 1) > 0$ **then**
15: $\quad\quad \rho(j) = c$
16: $\quad\quad b_j = \left\lceil \frac{j}{\mu_I} \right\rceil$

```
17:    for l = 1 to b_j do
18:     for m = 0 to μ_I − 1 do
19:      i = (l − 1) · μ_I + m
20:      if BP(j, l)&2^m == 1 then
21:       α(c) = 0
22:       κ_save(c) = κ(c) = i
23:       c = c + 1
24:      end if
25:     end for
26:    end for
27:   else
28:    ρ(j) = c
29:   end if
30:  end for
```

Algorithm 2.14 (Put$((\alpha, \kappa, \rho), \mathtt{j}, \mathtt{i}, \mathtt{c_{j,i}})$: Linear Partial Derivative Insertion)**.**

Require: (α, κ, ρ) after construction step in Algorithm 2.13 and the value of $c_{j,i}$.
Ensure: (α, κ, ρ) containing the partial derivative $c_{j,i}$.

```
1: l = Find((α, κ, ρ), j, i)
2: if l > 0 then
3:  if κ(l) == i then
4:   α(l) = c_{j,i}
5:  else
6:   α(l) = c_{j,i}
7:   κ(l) = i
8:  end if
9: end if
```

Algorithm 2.15 (SPut $(\mathtt{BP}, \mathtt{j}, \mathtt{i})$: Symbolic Nonzero Insertion)**.**

Require: BP and indices $i, j \in V$ with $i \prec j$.
Ensure: BP with additional entry on row j representing $c_{j,i}$.

```
1: k_1 = ⌈ i / μ_I ⌉
2: k_2 = ⌈ j / μ_I ⌉ + 1
3: e = (i − 1) % μ_I
4: if BP(j, k_1)&2^e == 0 then
5:  BP(j, k_1) = BP(j, k_1) + 2^e
6:  BP(j, k_2) = BP(j, k_2) + 1
7: end if
```

Algorithm 2.16 (ResetCRS$((\alpha, \kappa, \rho), \kappa_{\mathtt{save}})$: CRS Reset)**.**

Require: (α, κ, ρ), and initial column index vector κ_{save}.
Ensure: Reseted CRS to the initial state resulted by Algorithm 2.13;

```
1: for i = n + 1 to q do
```

2: **for** $l = \rho(i)$ **to** $\rho(i+1) - 1$ **do**
3: $\alpha(l) = 0$
4: $\kappa(l) = \kappa_{save}(l)$
5: **end for**
6: **end for**

Assumption 2.1. *The control flow of F is fix in $I \subseteq D$.*

In the following we focus our interest on the Jacobian of F at multiple points $\mathbf{x} \in I$, for which Assumption 2.1 holds. Hence, the Jacobian of F at any point in I can be accumulated on a static CRS resulting from a single symbolic step as described in the following.

Procedure 2.1. *The process of sparse Jacobian accumulation on a static compressed row storage as shown in Figure 2.11 can be summarized as follows. Here, the arrows correspond to the routine calls, whereas boxes represent the reached state after the routine call attached to the corresponding incoming arrows.*

- *(SYM)* **Symbolic Mode** *shown in the left column:*
  1. *BP is initialized during the evaluation of F at point* $\mathbf{x}$ *by calling* `SPut(BP,j,i)` *attached to arrow 1 for all $j = n+1, \ldots, q$ with $i \prec j$ as described in Algorithm 2.15.*
  2. *The filled bit pattern $\tilde{BP}$ is computed by calling* $\texttt{JSRowElim}(\texttt{BP}, \texttt{D}, \sigma)$ *attached to arrow 2.*
  3. *CRS is constructed by calling* $\texttt{ConstructCRS}(\tilde{\texttt{BP}}, (\alpha, \kappa, \rho), \kappa_{\texttt{save}})$ *attached to arrow 3.*
- *(ACC)* **Accumulation Mode** *shown in the right column:*
  1. *CRS is initialized by the evaluation of F at point* $\mathbf{x}$ *by calling* $\texttt{Put}((\alpha, \kappa, \rho), \texttt{j}, \texttt{i}, \texttt{c}_{\texttt{j,i}})$ *attached to arrow 4 for all $j = n+1, \ldots, q$ with $i \prec j$.*
  2. *The eliminated CRS $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho})$ is computed by calling* $\texttt{JRowElim}((\alpha, \kappa, \rho), \sigma)$ *attached to arrow 5.*
  3. *The Jacobian $\nabla F(\mathbf{x})$ is extracted from $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho})$ by calling* $\texttt{JExtract}((\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho}), \nabla\texttt{F})$ *attached to arrow 6.*
  4. *The steps 1-3 can be repeated to accumulate Jacobian at another point of interest after resetting the CRS to the initial state by calling* $\texttt{ResetCRS}\,((\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho}), \kappa_{\texttt{save}})$ *attached to arrow 7.*

2.4.2 Numerical Results

In the following we present numerical results on the entire process of Jacobian accumulation by row elimination on dense extended Jacobians (DJARE) as well as the respective CRS representations (SJARE). Henceforth, we will use CRS in our plots to denote the runtime and memory measurements for SJARE, which the sum of those of the symbolic and accumulation steps. Henceforth, we use the terminologies:

- DEJ to denote the dense extended Jacobian, and
- GFM and GRM to refer to the Jacobian accumulation on DEJ/CRS in forward and reverse ordering, respectively. Here, the entire DEJ/CRS is assumed to fit into the available memory. Otherwise, no Jacobian accumulation is possible in this mode. We refer to this case also as *non-iterative* mode.

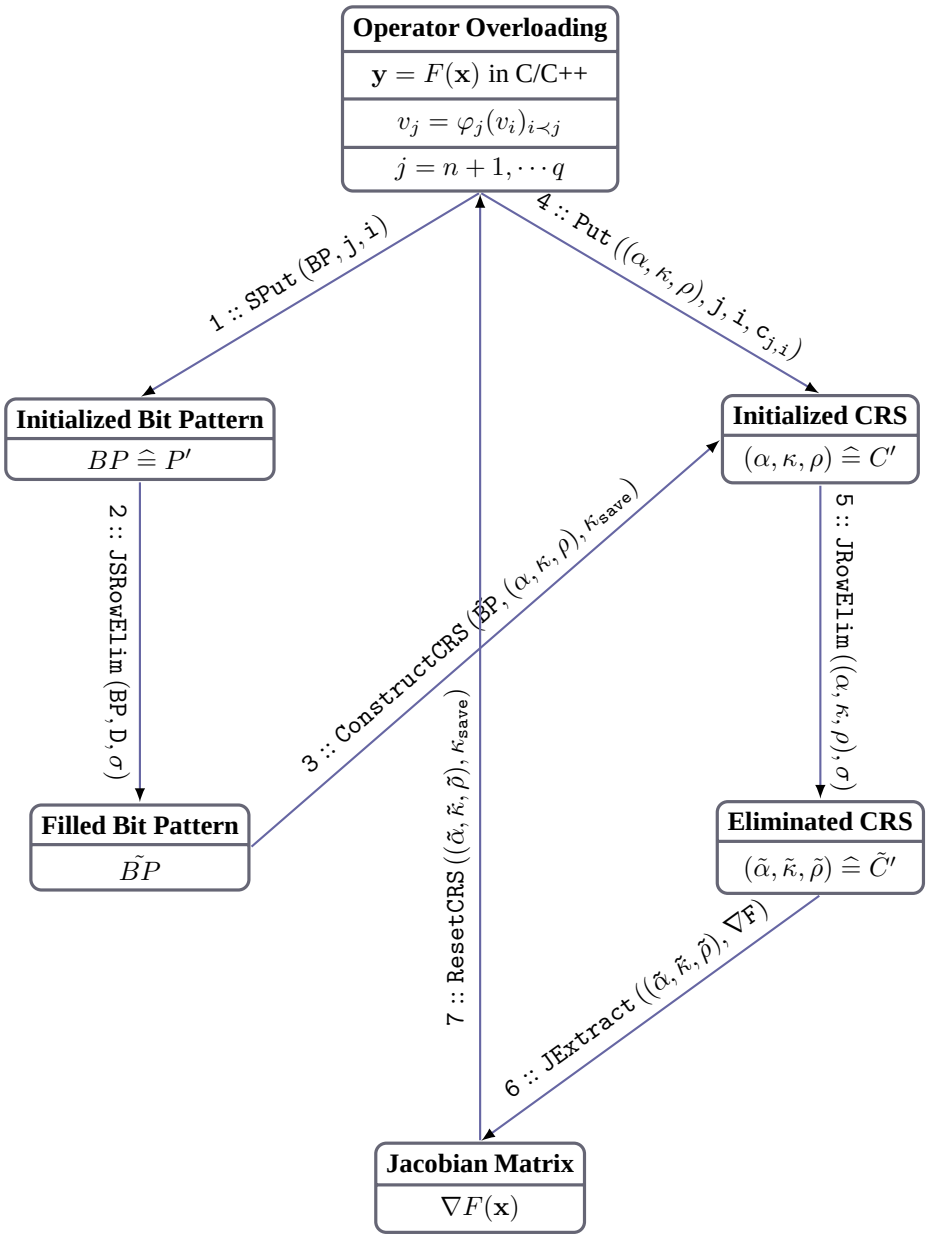


Figure 2.11: Process of Jacobian accumulation via Elimination of Rows on CRS in σ-order.

The tests are performed using the C++ operator overloading tool DALG attached to this work on an Intel Xeon X7460 @2.66GHz with 4 CPUs, 6 Cores per CPU, 3x3MByte L2-Cache, 16MByte L3-Cache, and totally 128 GByte RAM representing a node of the linux SMP cluster at Computing and Communication Center of the RWTH Aachen University.

The state-of-the-art implementation of DALG implements almost all algorithms and ideas illustrated in this chapter, except for those for reusing fill-out for fill-in as explained in Section 2.3. We emphasize that DALG stores almost all of its internal data structures such as DEJ and CRS on heap.

Bratu Problem

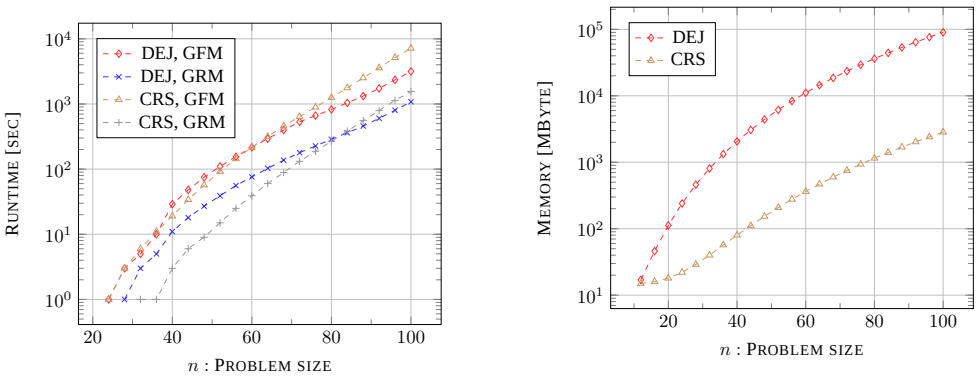
As first test case we consider an implementation of the two-dimensional Solid Fuel Ignition problem also known as the Bratu problem from MINPACK-2 test problem collection [ACM91]. As described by Naumann[Nau11], the residual function shown in Listing 2.1 is the result of replacing the differential

$$\Delta y \equiv \frac{\partial^2 y}{\partial x_0^2} + \frac{\partial^2 y}{\partial x_1^2}$$

in the elliptic partial differential equation

$$\Delta y - \lambda \cdot e^y = 0$$

with a set of algebraic equations using finite difference approximation as basic discretization method on the unit square $\Omega = [0, 1]^2$ denoting the boundary domain. The total runtime and heap memory behavior of DALG in non-iterative mode are shown in Figure 2.12 (a) and (b), respectively. The memory plot (b) indicates the maximum allocated heap memory during the entire Jacobian accumulation on DEJ resp. CRS for $n = 12, 16, \ldots, 100$, where the input $\mathbf{x}$ is a $n \times n$ floating point matrix. As one can see, the



(a) (b)

Figure 2.12: Runtime (a) and Memory (b) Behavior of DALG on Bratu in Non-Iterative Mode.

memory usage of DALG for computing the Jacobian of the dimension $(n^2 \times n^2)$ of the Bratu function is reduced drastically using CRS. The achieved gain on memory is about a factor of thirty-one for $n = 100$. More precisely, for $n = 100$ DEJ allocates roughly 90167 MByte of memory on heap, whereas CRS needs 2837 MByte for the same dimension. We note again that we duplicate bit pattern rows to keep right dependencies as well as total required memory as discussed in Section 2.4.1. Thus, the gain factor of thirty-one seems to be reasonable on our 64-bit test machine described above.

We note that the memory measurements are given as integer values in MByte. Moreover, the memory usage of CRS in both forward and reverse orderings are approximately the same in all of our experiments with Bratu. Hence, we present only memory usage of the former in Figure 2.12 (b). To clarify this, let us have a closer look at runtime and memory data of DALG of $n = 52$ as shown in Table 2.1. Here, forward and reverse elimination yield 88800 and 44800 fill-ins, respectively. Thus, the respective CRS of the former requires $44000 = 88800 - 44800$ more spots, that is, 704000 Bytes as

$$704000 = 44000 \times \mu_F + 44000 \times \mu_I \quad \text{for} \quad \mu_F = 8 \quad \text{and} \quad \mu_I = 8 \quad .$$

Hence, we get $0.67 \approx 704000/(1024)^2$ MByte, which is negligibly small. In an analogous manner, it holds for $n = 100$. Obviously, memory usage of DEJ is fixed for both orderings. As shown in (a), the

Elimination Mode	Time(DEJ)	Time(CRS)	#Muls	#Fill-in	#Entries	ω
Forward ($n = 52$)	110 sec.	92 sec.	93800	88800	141100	0.531259
Reverse ($n = 52$)	39 sec.	15 sec.	49800	44800	97100	0.0657922
Forward ($n = 100$)	3162 sec.	7159 sec.	362600	343392	544684	0.531108
Reverse ($n = 100$)	1082 sec.	1556 sec.	191688	172480	373772	0.0657927

Table 2.1: Summary of DALG Measurement Data for Bratu.

reverse elimination exhibits better runtime results on both DEJ and CRS. Thereby, non-iterative forward and reverse elimination show better runtime behavior on DEJ by increasing the dimension n than on their CRS counterpart. However, the former can not be used further to handle higher dimensions because of the memory bandwidth.

Therefore, let us consider again Table 2.1. As one can see, reverse elimination on DEJ is factor of roughly $2.8 \simeq \frac{110}{39}$ faster than forward one. This becomes clear when considering the multiplication ratio $1.9 \simeq \frac{93800}{49800}$. The same holds in fact for reverse elimination on CRS that is a factor of roughly $6.1 \simeq \frac{92}{15}$ faster than forward. We suspect the reason for this might be the better performance of the linear spot search routine `Find`$(\cdot)$ described in Algorithm 2.8 for reverse ordering with knowledge that the forward elimination yields a factor of roughly $1.45 \approx \frac{141100}{97100}$ more nonzeros than its counterpart. In order to analyze this, we compute the average linear search ratio

$$\omega = a \cdot \omega + \frac{(1-a) \cdot d}{l} \quad \text{with} \quad a = \frac{c-1}{c}$$

for every call of `Find`$(\cdot)$ on CRS with c denoting the current total number of calls. Thereby, l denotes the number of entries on considered rows, where on every row d indicates how many of its elements are considered until the algorithm terminates. For our example, this ratio is about 0.53 resp. 0.065 in case of forward resp. reverse elimination ordering. This shows that the linear spot search in the former is far inferior to in the latter. This example illustrates the importance of the spot search on CRS for performance of SJARE.

Listing 2.1: Bratu

```
1 void bratu(int n, double** x, double l) {
2   double h = 1./(n-1);
3   double r[n][n];
4   // enforce boundary condition
5   for (int i = 0; i < n; i++) {
6     x[i][0] = 0.;   x[i][n-1] = 0.;   x[0][i] = 0.;
7   }
8   for (int i = 0; i < n; i++)  x[n-1][i] = 1.;
```

```
9   // iterate over inner points
10  for (int i = 1; i < (n-1); i++) {
11    for (int j = 1; j < (n-1); j++) {
12      r[i][j] = 0. - ((x[i+1][j] - 2 * x[i][j] + x[i-1][j]) / (h*h))
13      - ((x[i][j+1] - 2 * x[i][j] + x[i][j-1]) / (h*h))
14      - l * exp(x[i][j]);
15    }
16  }
17  // updating the inner points
18  for (int i = 1; i < n-1; i++)
19    for (int j = 1; j < n-1; j++)
20      x[i][j] = r[i][j];
21 }
```

In conclusion, it is worth mentioning that the symbolic row elimination for $n = 52$ needs totally 10 seconds to predict the memory requirement for the following accumulation step, which takes only 5 seconds. Hence, the latter seems to perform twice better than the former. We note that this behavior is also observed for the following problem.

Heat Equation

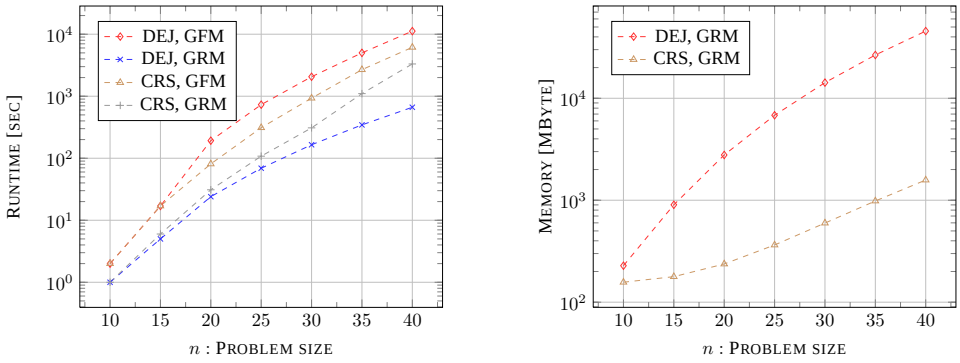
As second test case we consider the objective function $f : \mathbb{R}^{nx} \to \mathbb{R}$ implemented in lines 20-26 of Listing 2.2. Our objective is to accumulate the gradient ∇f needed in context of a steepest descent algorithm minimizing the difference between the initial temperature (condition) T^0 and the distributed simulated temperature $T^{nt} = F(T^0) : \mathbb{R}^{nx} \to \mathbb{R}^{nx}$ after nt time steps of a simple integration of the one-dimensional *heat equation* [Hea97]. A bar of given length is heated on one side for some time. The simulated temperature distribution is returned at a number of discrete points denoted by nx. The Heat problem is a linear ill-posed inverse problem. The routine `time_integration` in line 9 of Listing 2.2 shows a C++ implementation of T^{nt}.

Figure 2.13 compares both the runtime and memory behavior of DALG using DEJ and CRS for $nx = 10, 15, \ldots, 40$ with $nt = 10 \cdot nx$. Analogous to the Bratu case DEJ hits the memory bound much faster than CRS. The respective runtime behavior of both forward and reverse is very similar to that of Bratu as discussed previously.

Listing 2.2: heat

```
1  // single time step
2  void single_ts(int nx, double delta_t, double c,
3  double *temp, double *temp_new) {
4    for (int j = 1; j < nx; j++)
5      temp_new[j] = temp[j] +
6      c*nx*nx*delta_t*(temp[j+1]-2*temp[j]+temp[j-1]);
7  }
8  // time stepping scheme
9  void time_integration(int nx, int nt, double delta_t,
10 double c, double *temp) {
11   double *temp_new = new double[nx+1];
12   // time integration
13   for (int i = 0; i < nt; i++) {
14     single_ts(nx, delta_t, c, temp, temp_new);
15     for (int j = 0; j < nx+1; j++)  temp[j] = temp_new[j];
16   }
17   delete [] temp_new;
```

```
18 }
19 // Objective function
20 void f(int nx, int nt, double delta_t, double c,
21 double *temp, double *temp_obs, double &cost) {
22   time_integration(nx,nt,delta_t,c,temp);
23   cost = 0.0;
24   for (int j = 0; j <= nx; j++)
25     cost += (temp[j]-temp_obs[j])*(temp[j]-temp_obs[j]);
26 }
```



(a) (b)

Figure 2.13: Runtime (a) and Memory (b) Usage of DALG on Heat in Non-Iterative Mode.

As summarized in Procedure 2.1 the CRS can be reused to accumulate Jacobians of one and the same function at different points, assuming the unchangeability of the control flow of the target function. The function f represents exactly such a function. Figure 2.14 presents the mean runtime

$$g(i) = \frac{\sum_{j=1}^{i} t_j}{i}$$

for computing the gradient ∇f of f in the context of the steepest descent algorithm mentioned above. Therefore, we manage to compute ∇f in reverse mode on DEJ resp. CRS at every iteration i of the algorithm for $nx = 40$ and $nt = 400$. As one can see the overhead of the symbolic step on CRS is compensated in the accumulation step as proceeding with iterations, whereas DEJ behaves almost consistently.

In conclusion with respect to our numerical results, we observed that DEJ tends to hit the memory limit very quickly. Here CRS can be used to yield better scalability by exploiting the sparsity of DEJ, which improves the memory consumption substantially. However, the reader may agree that even the capability of the SJARE is limited by the memory consumption of the bit pattern. Section 2.6 will present our idea for handling this problem. Nonetheless, in both considered test cases we also observed that the gain in runtime on CRS gets asymptotically smaller compared with DEJ when increasing the dimension of both problems. We conjecture the reason to lie in the fact that increase in n results in larger search space for dependencies (larger q) as shown in line 1 of Algorithm 2.2 and line 1 of Algorithm 2.6 on DEJ and CRS, respectively. Moreover, we note that on CRS finding the dependency of a particular row on another as well as finding a spot for an entry is done with a linear overhead, whereas DEJ needs O(1) in both cases. Hence, the performance of SJARE seems to depend very much on the efficiency of the spot search and **the size of the search space** as well. The impact of the latter on the performance of SJARE

becomes more clear in context of parallel Jacobian accumulation in non-iterative fashion as introduced below.

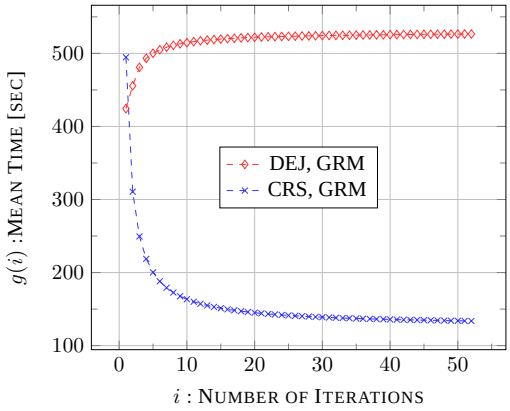


Figure 2.14: Mean Time of DALG on Heat using DEJ resp. CRS for $nx = 40$ and $nt = 400$.

2.5 Parallel Jacobian Accumulation

The focus in the following is on finding approaches to parallelizing the Jacobian accumulation process discussed in the previous chapter, which we refer to as *parallel Jacobian accumulation* (PJA) [VN07]. For simplicity, ideas are illustrated on G of F defined by Equation (1.13). However, the correspondence between DAG and both internal representations DEJ and CRS used here along with the respective elimination algorithms have been explained in detail previously.

Thus, we introduce in the following two ideas for parallelizing vertex elimination on G. We still assume that G fits entirely into the available memory. Thus, elimination of all intermediate vertices Z in serial fashion yields the bipartite graph $\tilde{G} = G - Z$ with edge labels representing the entries of $\nabla F(\mathbf{x})$ as discussed in Section 1.2. To support the discussion below and to address issues related to the parallelization of vertex elimination let us consider G of Figure 2.15 with vertices

$$V = \{1, \ldots, 14\}, \quad \text{where} \quad X = \{1, \ldots, 6\}, \quad Z = \{7, 8, \ldots, 13\}, \quad \text{and} \quad Y = \{14\} \quad .$$

Clearly, the elimination of intermediate vertices yields the complete bipartite graph $\tilde{G}$ with X and Y as source and target vertices, respectively. Let us consider now two disjoint decompositions $Z_1 = \{7, 8, 9, 10\}$ and $Z_2 = \{11, 12, 13\}$ of Z representing two *vertex decompositions* of G. Obviously, Z_1 and Z_2 can be eliminated simultaneously for instance by processes P_1 and P_2 as there is no mutual dependency among their vertices. The resulting DAG after the parallel elimination process is given by $\tilde{G} = G - (Z_1 \cup Z_2)$.

Let us now consider G in Figure 2.16 (a), which is a modified version of G in Figure 2.15, where Z_1 and Z_2 are not independent anymore because of the edge $(9, 12)$. We refer to such an edge connecting vertices of two different decompositions of Z as *out-of-range*. Now, let us assume that P_1 and P_2 still try to eliminate vertices 9 and 10 in parallel. Thereby, it can happen that $(9, 12)$ is accessed in *read and write* fashion by P_1 and P_2, respectively or vice versa. This is a typical case of *data race*, where the chain rule correctness can not be guaranteed anymore. For instance, while back-eliminating $(9, 12)$ of 9 by P_1 process P_2 may access $(9, 12)$ to get the value of the local partial derivative attached to it, which is needed for back-eliminating $(12, 13)$.

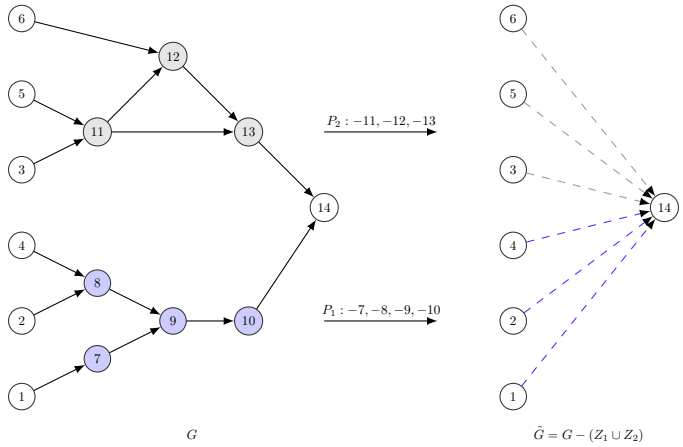


Figure 2.15: Parallel Vertex Elimination with no Communication.

One way to solve this problem is to have the concerned processes P_1 and P_2 communicate with each other. For instance, P_1 eliminates vertex 9 while P_2 waits. Thereby, fill-in $(7, 10)$, $(8, 10)$, $(7, 12)$, and $(8, 12)$ are generated as shown in $G - 9$. We note that the communication here doesn't have to be a blocking one as P_2 can eliminate 11 and 13 while waiting for a signal from P_1 in order to eliminate 12 as well. Unfortunately, eliminating vertex 9 not really reduces the communication cost as fill-ins $(7, 12)$ and $(8, 12)$ are also out-of-range ones; this means even further communication between P_1 and P_2, which may slow down PJA significantly. The easiest way to circumvent this problem is to avoid the elimination of all those intermediate vertices incident to out-of-range edges that we refer to as *critical vertices*. Nonetheless, this may decrease, on the other hand, the number of eliminatable vertices of the respective decompositions and thus affect the load balancing. For our example, this could result in $Z_1 = \{7, 8, 10\}$ and $Z_2 = \{11, 13\}$ yielding $G - \{7, 8, 10, 11, 13\}$. Thereby, P_2 eliminates two vertices, whereas P_1 does eliminate three. Thus, out of the total of seven intermediates two remains in $G - \{7, 8, 9, 11, 13\}$ and five are eliminated. Hence, a further elimination step is needed to yield $\tilde{G}$. However, in practice, it is very likely that multiple levels of parallel vertex elimination sessions are needed on the way to $\tilde{G}$ as discussed below.

2.5.1 Atomic Decomposition

Due to the problem related to out-of-range edges discussed above keeping their number minimal is an important and a more challenging task. In this step the main focus is on having balanced decompositions [MK08, CP08] to optimize the computational and communication cost in concurrent processes. However, in the following we assume the decompositions to be the result of user-driven (hard-wired) code instrumentation marking parallel fragments of F, which we assume to be at the loop level. An example instrumentation is given in lines 16-19 of Listing 2.4. More detail on this is discussed in Section 2.5.4. Nonetheless, in general we are looking for a decomposition of G into ν *atomic subgraphs* defined as follows.

Definition 2.4. *Given DAG $G = (V, E)$ of F as defined by Equation (*1.13*) with topologically ordered vertices V. We say G is* atomically decomposable *if there exist ν subgraphs*

$$G_i = (V_i, E_i) \quad \textit{with} \quad V \supseteq V_i = (X_i \cup Y_i \cup Z_i) \quad \textit{and} \quad E_i \subseteq E \,, \tag{2.11}$$

where $E_i \cap E_j = \emptyset$ and $Z_i \cap Z_j = \emptyset$ for $i, j \in \{1, \cdots, \nu\}$ such that $V = \bigcup_{i=1}^{\nu} V_i$ and $E = \bigcup_{i=1}^{\nu} E_i$.

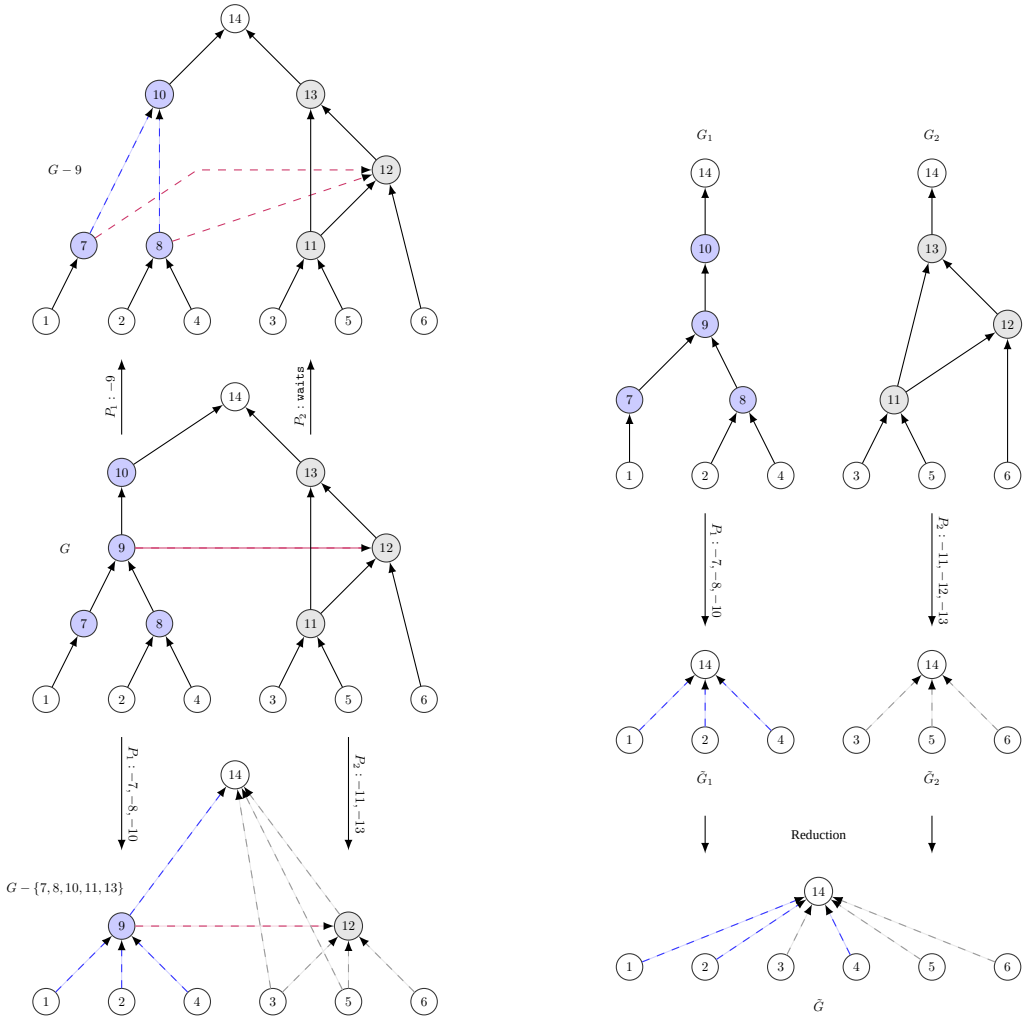


Figure 2.16: Parallel Jacobian Accumulation with Communication (a) and Reduction (b), respectively.

Strictly speaking, a subgraph is atomic if all of its edges are among vertices of that subgraph. Moreover, X_i and Y_i represent the local independent and local dependent vertices of the subgraph G_i, respectively. Furthermore, $Z_i = V_i - (X_i \cup Y_i)$ represents the set of intermediate vertices of G_i that can be locally eliminated. Thus, X_i, Y_i, and Z_i are mutually disjoint, whereby their vertices are supposed to be mutually independent as well. We call subgraphs G_i and G_j neighbors for $j = i + 1$.

Hence, we get the atomic subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as shown in Figure 2.16 (b) for G of Figure 2.15, where

$$\begin{aligned} X_1 &= \{1, 2, 4\}, \quad Z_1 = \{7, 8, 9, 10\}, \quad Y_1 = \{14\}, \\ X_2 &= \{3, 5, 6\}, \quad Z_2 = \{11, 12, 13\}, \quad \text{and} \quad Y_2 = \{14\} \quad . \end{aligned}$$

Here, the vertex 14 is the common vertex of both subgraphs G_1 and G_2. However, this does not affect the parallelization in terms of data race during the elimination process as it is not eliminated by any of the processes P_1 and P_2. Hence, eliminating vertices Z_1 and Z_2 in parallel by P_1 and P_2 yields *local bipartite graphs* $\tilde{G}_1$ and $\tilde{G}_2$, respectively. Finally, $\tilde{G}_1$ and $\tilde{G}_2$ are reduced to $\tilde{G}$. The reduction step depends very much on the type of involved atomic subgraphs that will be discussed below.

Under the assumption that G of F can be atomically decomposed into ν atomic subgraphs $G_i = (V_i, E_i)$ with $i \in \{1, \ldots, \nu\}$, F can be considered as a composition of ν functions

$$F_i : \mathbb{R}^{n_i} \to \mathbb{R}^{m_i} \quad : \quad \mathbf{w} = F_i(\mathbf{v}) \quad \text{with} \quad \mathbf{v} \equiv X_i \quad \text{and} \quad \mathbf{w} \equiv Y_i$$

with $n_i = |X_i|$, $m_i = |Y_i|$ such that $X \subseteq \bigcup_{i=1}^{\nu} X_i$ and $Y \subseteq \bigcup_{i=1}^{\nu} Y_i$. The notation $\mathbf{v} = (v_1, \cdots, v_{n_i}) \equiv X_i$ [$\mathbf{w} = (w_1, \cdots, w_{m_i}) \equiv Y_i$] denotes that the vector elements of $\mathbf{v}$ [$\mathbf{w}$] are represented by vertices X_i [Y_i] of G_i. Thus, we get the local bipartite graph

$$\tilde{G}_i = (\tilde{V}_i, \tilde{E}_i) = G_i - Z_i$$

by eliminating its intermediate vertices Z_i. This corresponds to the local application of Baur's formula defined by Equation (1.16) yielding each entry

$$f'_{i,j,l} = \frac{\partial w_j}{\partial v_l}(\mathbf{v}) = \sum_{\pi \in \{l \to j\}} \prod_{(p,k) \in \pi} c_{k,p}$$

of the *local Jacobian*

$$(\mathbb{R}^{m_i \times n_i} \ni) \ \nabla F_i = \nabla F_i(\mathbf{v}) \equiv \left(f'_{i,j,l}\right)_{l=1,\ldots,n_i}^{j=1,\ldots,m_i}$$

as the elimination of all paths π connecting an independent vertex $l \in X_i$ to a dependent one $j \in Y_i$. Hence, $\tilde{G}$ can be obtained from $\tilde{G}_i$ in the reduction step. Here, we consider four types of atomic decompositions as illustrated in Figures 2.17. However, the reduction is performed in general by first combining multiple local bipartite graphs to a *composition graph* according to Definition (2.5) followed by eliminating the resulting interface vertices, yielding the respective bipartite graphs as illustrated by CASE 4.

Definition 2.5. *Given atomic DAGs $G_i = (V_i, E_i)$ defined by Equation (2.11) for $i \in M = \{1, \ldots, \nu\}$ with sets of independent X_i, dependent Y_i and intermediate vertices Z_i. The notation*

$$G(S) = (V(S), E(S)) \equiv (G_j)_{j \in S} \tag{2.14}$$

with $S \subseteq M$ denotes the composition DAG consisting of $|S|$ atomics G_j with $j \in S$ and $V(S) = \bigcup_{j \in S} V_j$

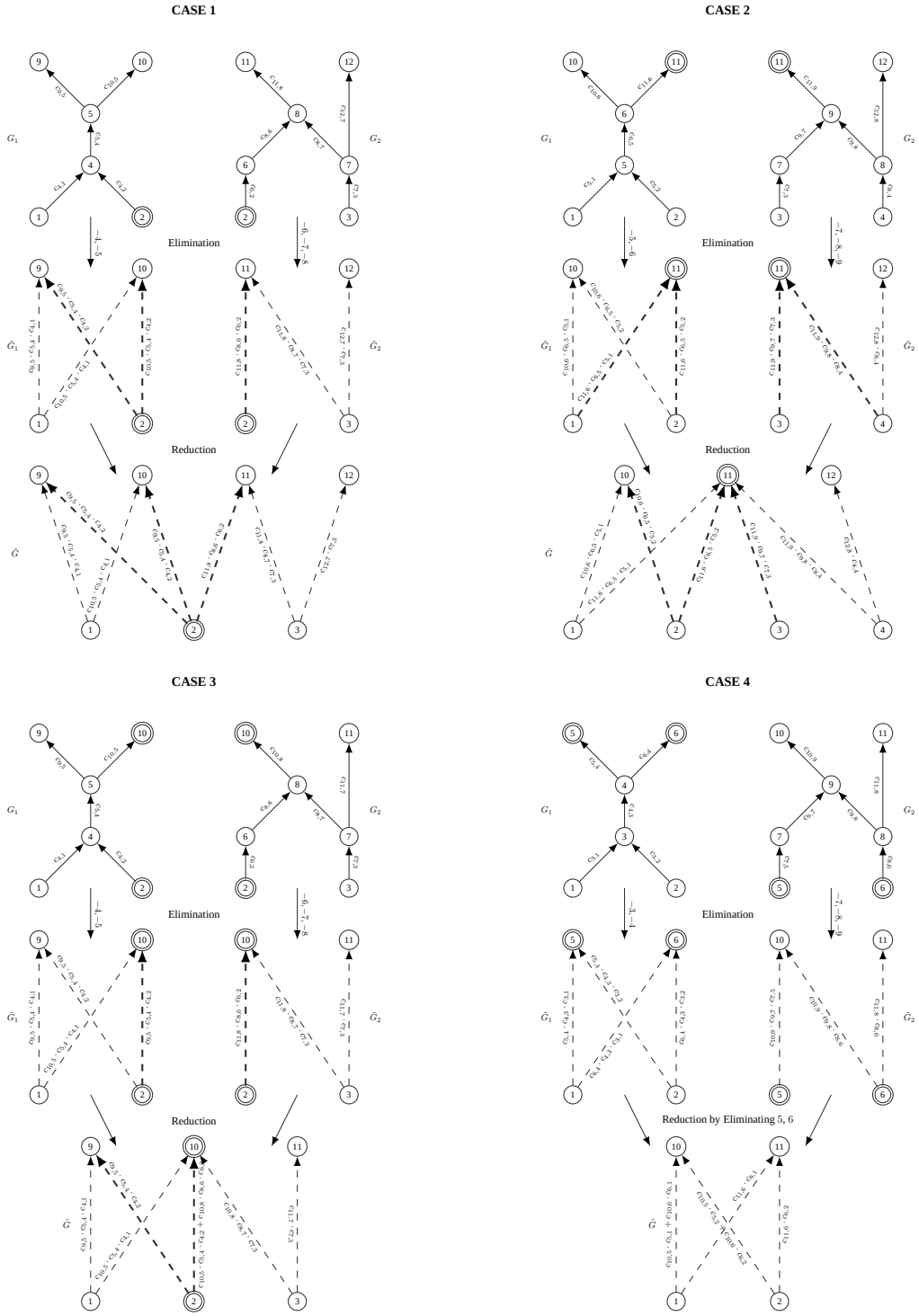


Figure 2.17: Reduction Step for Two Atomic Subgraphs sharing Interface Vertices.

and $\bigcup_{j \in S} E_j$ *such that*

$$X(S) = \bigcup_{j \in S} X_j - Interface(S), \quad Y(S) = \bigcup_{j \in S} Y_j - Interface(S), \quad \textit{and}$$

$$Z(S) = \bigcup_{j \in S} Z_j \cup Interface(S) \quad \textit{with} \quad Interface(S) = \bigcup_{i \neq j \in S} (X_i \cap Y_j) \cup (X_j \cap Y_i) \quad .$$

As an example let us consider G in Figure 2.18 (a) with

$$X = \{1, 2, 3\}, \quad Z = \{4, 5, 6, \ldots, 19\}, \quad \text{and} \quad Y = \{20, 21\}$$

that is decomposed into totally four atomic subgraphs G_1, G_2, G_3, and G_4 as

$$\begin{aligned}
&X_1 = \{1, 2, 3\}; \quad Z_1 = \{4, 5\}; \quad Y_1 = \{6, 7\}, \\
&X_2 = \{6, 7\}; \quad Z_2 = \{8, 9\}; \quad Y_2 = \{10, 11, 12\}, \\
&X_3 = \{10, 11, 12\}; \quad Z_3 = \{13, 14\}; \quad Y_3 = \{15, 16, 17\}, \quad \text{and} \\
&X_4 = \{15, 16, 17\}; \quad Z_4 = \{18, 19\}; \quad Y_4 = \{20, 21\} \quad .
\end{aligned}$$

Thus, application of the vertex elimination locally to the subgraphs G_i for $i = 1 \ldots, 4$ of G yields $\tilde{G}_i$ with edge labels representing the entries of ∇F_i. As one can see, our example DAG has the property that dependents of a subgraph G_i serve as independents of its next neighbor G_{i+1}, that is, $Y_i = X_{i+1}$. Such a decomposition will be the main focus in the following. Obviously, the functions F_i and F_j with $j = i + 1$ corresponding to G_i and G_j have the property that the outputs of F_i serve as inputs for F_j for $i \in \{1, \ldots, \nu\}$ with $\mathbf{x}_1 = \mathbf{x}$ and $\mathbf{x}_j = \mathbf{y}_i$ such that F can be represented as the chain

$$\mathbf{y} = F(\mathbf{x}) = F_\nu \circ \ldots \circ F_1(\mathbf{x}) \quad .$$

Thereby, G_i and G_j share Y_i as their interface vertices. Moreover, we have

$$X_j = Y_i, \quad X_i \cap X_j = \emptyset, \quad \text{and} \quad Z_i \cap Z_j = \emptyset \quad .$$

Consequently, the Jacobian ∇F can be computed as chained product

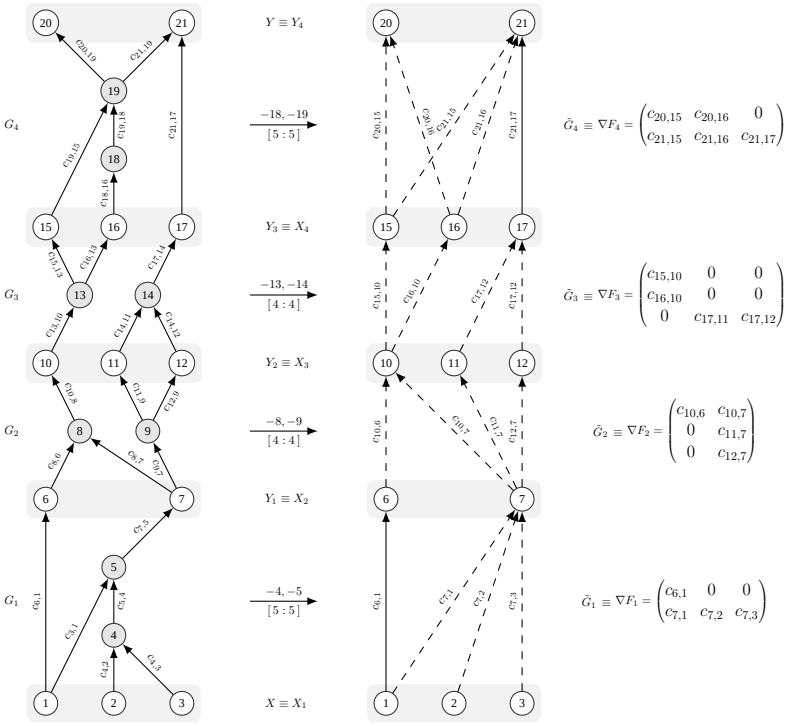
$$\nabla F = \nabla F_\nu \times \cdots \times \nabla F_1$$

of local Jacobians ∇F_i. Hence, dynamic programming [GN03] (DP) can be used to optimize the number of performed multiplications (MULS) by finding an optimal bracketing scheme. As shown in Figure 2.18 (b) the dense chained matrix product applied to our example yields the following two optimal bracketings

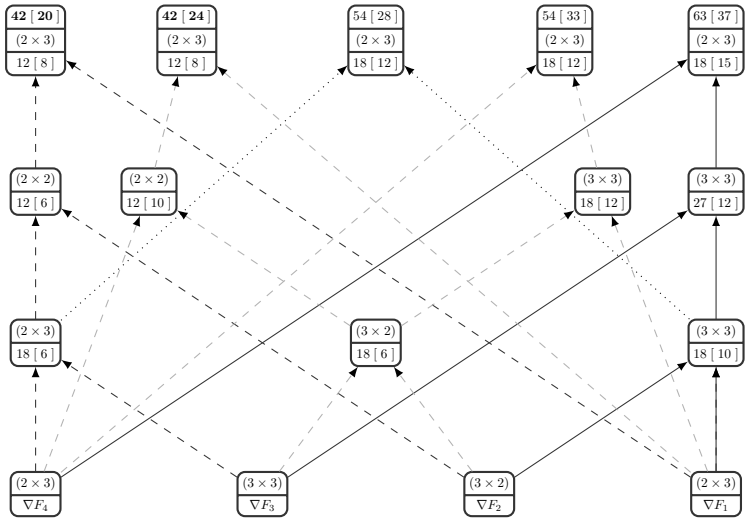
$$\nabla F_4 \times (\nabla F_3 \times (\nabla F_2 \times \nabla F_1)) \quad \text{and} \quad \nabla F_4 \times ((\nabla F_3 \times \nabla F_2) \times \nabla F_1))$$

resulting in 42 MULS, instead of 54, 54, and 63 MULS for the three remaining ones. The application of the sparse method to the former yields 20 MULS, instead of 24, 28, 33, and 37 MULS for the four others. It is worth mentioning here that the product of two local Jacobians ∇F_i and ∇F_j with $j = i + 1$ can be interpreted graphically as back-elimination of all inedges of vertices in Y_j resulting in their elimination from G. For instance, vertices 15, 16, and 17 in Figure 2.19 are removed as a consequence of back-eliminating all inedges of 20 and 21. Thereby, the product of the first row of ∇F_4 with first column of ∇F_3 realizes the back-elimination of $(15, 20)$ and $(16, 21)$.

We note that the order in which vertices of an interface are eliminated is not important as they are assumed to be mutually disjoint. This means that any elimination ordering yields the same number of



(a)



(b)

Figure 2.18: Vertex Elimination on Atomic Subgraphs (a) and possible Bracketing Schemes (b) for the resulting local Jacobians. Entries $[a : b]$ below of Arrows in the Former denote the resulting Number of Multiplications (a) resp. Fill-in (b).

multiplications as well as fill-ins. However, this is not true in general, especially when we reduce more than two local bipartite graphs, that is, reducing vertices of multiple interfaces at once.

Henceforth, we do not distinguish between a given bracketing scheme and the resulting vertex elimination. The optimal sparse chained product of concerned matrices (local Jacobians) of Figure 2.18 (a) are exercised in Example 2.6. The respective vertex elimination is shown in Figure 2.19.

Example 2.6. *In the following we apply the optimal bracketing scheme*

$$((\nabla F_4 \cdot \nabla F_3) \cdot \nabla F_2) \cdot \nabla F_1$$

resulted from DP to the local extended Jacobians ∇F_i for $i = 1, 2, 3, 4$ corresponding to the local bipartite graphs $\tilde{G}_i$ as shown in Figure 2.18 (a). Thereby, we show the correspondence to vertex elimination in Figure 2.19. For this, we consider G_5 in (a) consisting of $\tilde{G}_4$ and $\tilde{G}_3$ such that

$$X_5 = \{10, 11, 12\}, \quad Z_5 = \{15, 16, 17\}, \quad \textit{and} \quad Y_5 = \{20, 21\} \quad .$$

Hence, computing the product

$$\begin{aligned}\nabla F_5 &= \nabla F_4 \cdot \nabla F_3 \\ &= \begin{pmatrix} c_{20,15} & c_{20,16} & 0 \\ c_{21,15} & c_{21,16} & c_{21,17} \end{pmatrix} \cdot \begin{pmatrix} c_{15,10} & 0 & 0 \\ c_{16,10} & 0 & 0 \\ 0 & c_{17,11} & c_{17,12} \end{pmatrix} = \begin{pmatrix} \mathbf{c_{20,10}} & 0 & 0 \\ \mathbf{c_{21,10}} & \mathbf{c_{21,11}} & \mathbf{c_{21,12}} \end{pmatrix}\end{aligned}$$

corresponds to the elimination of vertices Z_5 at a cost of six MULS, where

$$\begin{aligned}&\mathbf{c_{20,10}} = c_{20,15} \cdot c_{15,10} + c_{20,16} \cdot c_{16,10}, \quad \mathbf{c_{21,10}} = c_{21,15} \cdot c_{15,10} + c_{21,16} \cdot c_{16,10}, \\ &\mathbf{c_{21,11}} = c_{21,17} \cdot c_{17,11}, \quad \textit{and} \quad \mathbf{c_{21,12}} = c_{21,17} \cdot c_{17,12}\end{aligned}$$

represent the labels of the fill-in edges $(10, 20)$, $(10, 21)$, $(11, 21)$, and $(12, 21)$ in (b), respectively. Thereby, the product of the first row of ∇F_4 with first column of ∇F_3 results in the elimination of (15,20) and (16,20). Likewise, the elimination of vertices Z_6 of G_6 with

$$X_6 = \{6, 7\}, \quad Z_6 = \{10, 11, 12\}, \quad \textit{and} \quad Y_6 = \{20, 21\}$$

cab be interpreted as the product of

$$\nabla F_6 = \nabla F_5 \cdot \nabla F_2 = \begin{pmatrix} c_{20,10} & 0 & 0 \\ c_{21,10} & c_{21,11} & c_{21,12} \end{pmatrix} \cdot \begin{pmatrix} c_{10,6} & c_{10,7} \\ 0 & c_{11,7} \\ 0 & c_{12,7} \end{pmatrix} = \begin{pmatrix} \mathbf{c_{20,6}} & \mathbf{c_{20,7}} \\ \mathbf{c_{21,6}} & \mathbf{c_{21,7}} \end{pmatrix}$$

yielding six MULS with

$$\begin{aligned}&\mathbf{c_{20,6}} = c_{20,10} \cdot c_{10,6}, \quad \mathbf{c_{20,7}} = c_{20,10} \cdot c_{10,7}, \quad \mathbf{c_{21,6}} = c_{21,10} \cdot c_{10,6}, \quad \textit{and} \\ &\mathbf{c_{21,7}} = c_{21,10} \cdot c_{10,7} + c_{21,11} \cdot c_{11,7} + c_{21,12} \cdot c_{12,7} \quad .\end{aligned}$$

representing the labels of $(6, 20)$, $(6, 21)$, $(7, 20)$, and $(7, 21)$ in (c), respectively. Finally, we get the entire bipartite graph $\tilde{G}$ shown in (d) by eliminating the intermediate vertices $Z_7 = \{6, 7\}$ with $X_7 = X_1$ and $Y_7 = Y_6$ at a cost of eight MULS as

$$\begin{aligned}&\mathbf{c_{20,1}} = c_{20,6} \cdot c_{6,1} + c_{20,7} \cdot c_{7,1}, \quad \mathbf{c_{20,2}} = c_{20,7} \cdot c_{7,2}, \quad \mathbf{c_{20,3}} = c_{20,7} \cdot c_{7,3}, \\ &\mathbf{c_{21,1}} = c_{21,6} \cdot c_{6,1} + c_{21,7} \cdot c_{7,1}, \quad \mathbf{c_{21,2}} = c_{21,7} \cdot c_{7,2}, \quad \textit{and} \quad \mathbf{c_{21,3}} = c_{21,7} \cdot c_{7,3} \quad .\end{aligned}$$

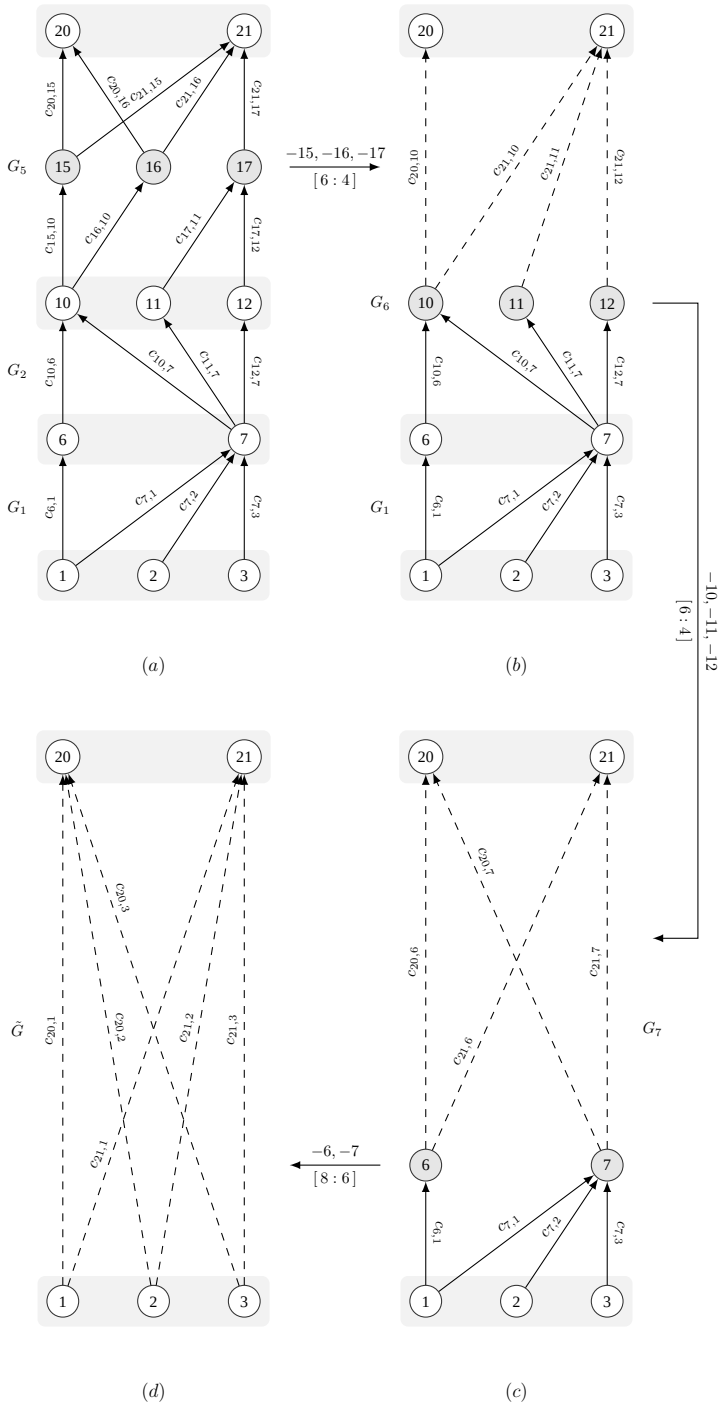


Figure 2.19: Vertex Elimination corresponding to the optimal Bracketing Scheme.

Elimination Mode	**MULS**	**Fill-in**
GFM	57	47
GRM	38	29
LFM	55	42
LRM	36	31
LFM+DP	38	32
LRM+DP	37	31

Table 2.2: Multiplication and Fill-in Comparison.

These represent the entries of the target Jacobian

$$\nabla F = \nabla F_7 = \nabla F_6 \cdot \nabla F_1 = \begin{pmatrix} c_{20,6} & c_{20,7} \\ c_{21,6} & c_{21,7} \end{pmatrix} \cdot \begin{pmatrix} c_{6,1} & 0 & 0 \\ c_{7,1} & c_{7,2} & c_{7,3} \end{pmatrix} = \begin{pmatrix} \mathbf{c_{20,1}} & \mathbf{c_{20,2}} & \mathbf{c_{20,3}} \\ \mathbf{c_{21,1}} & \mathbf{c_{21,2}} & \mathbf{c_{21,3}} \end{pmatrix} .$$

Hence, the Jacobian is computed at an optimal cost of totally twenty MULS.

To summarize and complete the discussion related to vertex elimination on atomic subgraphs we consider Table 2.2, which compares the resulting number of multiplications and fill-ins of different elimination orderings on G of Figure 2.18 (a). Here, GFM and GRM mean that the vertices Z of G are eliminated as usual in forward and reverse fashion. LFM [LRM] indicates local elimination of intermediates on subgraphs G_i for $i \in S = \{1, \ldots, 4\}$ in forward [reverse] ordering followed by eliminating the entire intermediates $Z(S)$ of the remaining DAG $G(S) \equiv \left(\tilde{G}_1, \ldots, \tilde{G}_4\right)$ with

$$X(S) = X_1, \quad Z(S) = \{6, 7, 10, 11, 12, 15, 16, 17\}, \quad \text{and} \quad Y(S) = Y_4$$

as shown in Figure 2.19 (a) at once in forward [reverse] order. Moreover, LFM+DP [LRM+DP] eliminates Z_i of G_i for $i = 1, \ldots, 7$ according to Equation (2.14) in forward [reverse] order consecutively, where the decomposition

$$G_5 \equiv G(\{3, 4\}), \quad G_6 \equiv (\{2, 5\}), \quad \text{and} \quad G = G_7 \equiv (\{1, 6\})$$

results from the optimal bracketing scheme using DP as discussed in Example 2.6.

For this example, we observe that GRM and LRM yield the smallest number of fill-in and multiplications, respectively. However, the decomposition of the DAG and local elimination on subgraphs seem to improve both operation and memory usage of global forward ordering. In the reverse case the former is improved as well, whereas the latter gets close to the respective global version. We note that in the case of LFM [LRM] we [would] get 18 [17] for the number of multiplications as well as for fill-ins [4] as shown in Figure 2.18 (a). Hence, the remaining number of 37 [19] multiplications and 24 [14] fill-ins result from the elimination of $Z(S)$ vertices in forward [reverse] order.

2.5.2 Pyramid Approach

In the following we present our fist idea for parallelizing the vertex elimination on atomic subgraphs. Here, the *pyramid approach* realizes a level-based parallel vertex elimination that is described in Listing 2.3.

Listing 2.3: Pyramid Algorithm

```
1 d = ⌈log_β N⌉ ;
```

[4] We left the calculation of both multiplication and fill-in numbers for LRM to the reader.

```
2  // Sessions
3  for l = 0, ..., d
4    if (l == 0)
5      N_l = N
6      for j = 1, ..., N_l
7        G_j^l = G_j
8    else
9      N_l = ⌈N_{l-1} / β⌉
10     // Decomposition Step
11     for j = 1, ..., N_l
12       G_j^l ≡ (G̃_{(j-1)·β+1}^{l-1}, ..., G̃_{j·β}^{l-1})
13   // Elimination Step
14   for j = 1, ..., N_l
15     G̃_j^l = G_j^l − Z_j^l
```

Therefore, we assume that N and β are given, where

- N denotes the number of initially atomic subgraphs in G and
- β represents the maximum number of atomics that can be combined together at the decomposition step as shown in lines 11-12.

We illustrate the pyramid algorithm on G given in Figure 2.20 for $N = 7$ and $\beta = 3$. At the lowest level $l = 0$ (lines 4-7) the computational graph G consists of seven ($N = N_0 = 7$) atomic subgraphs $G_1^0 = G_1, \ldots, G_7^0 = G_7$. Henceforth, we use the terminology *session* to refer to a level $l \in \{0, \ldots, d\}$ of pyramid algorithm with

$$d = \lceil \log_\beta N \rceil \tag{2.15}$$

denoting the maximum number of sessions being two ($d = 2$) for our example. In general, a session consists of a decomposition followed by an elimination step, where for $l = 0$ the decomposition step is not performed as G is assumed to be initially decomposed. Thus, decomposition at session $l > 0$ yields

$$G_j^l \equiv \left(\tilde{G}_i^{l-1}\right)_{i \in S} \quad \text{with} \quad S = \{(j-1) \cdot \beta + 1, \ldots, j \cdot \beta\}$$

for $j = 1, \ldots, N_l$ with N_l denoting the number of subgraphs at session l as shown in line 9. In other words, the decomposition at the level l is nothing else than building the composition of DAG according to Definition (2.5) out of β consecutive (neighboring) eliminated subgraphs resulting from the previous session $l - 1$. Obviously, only the elimination steps of sessions $l = 0, \ldots, d - 1$ can run in parallel with a maximum number of N_l processes, which decreases by increase in l.

We note here that the execution of the pyramid algorithm can be visualized as a $\beta-$ary tree [Sto01] with subgraphs denoting tree nodes that represent jobs has to be done by processes. One can easily figure out that d in Equation (2.15) denotes exactly the depth of this $\beta-$ary tree. An example is given for $\beta = 3$ in Figure 2.20.

Thus, applying the vertex elimination as described in lines 14-15 to all seven subgraphs in parallel using three processes P_1, P_2 and P_3 yields $\tilde{G}_1^0, \ldots, \tilde{G}_7^0$ representing the eliminated DAG $G^0 = G - \bigcup_{i=1}^7 Z_i^0$ at session $l = 0$. The session $l = 1$ starts first with decomposition of G^0 into three subgraphs

$$G_1^1 \equiv \left(\tilde{G}_1^0, \tilde{G}_2^0, \tilde{G}_3^0\right), \quad G_2^1 \equiv \left(\tilde{G}_4^0, \tilde{G}_5^0, \tilde{G}_6^0\right), \quad \text{and} \quad G_3^1 \equiv \left(\tilde{G}_7^0\right) \quad \text{with}$$

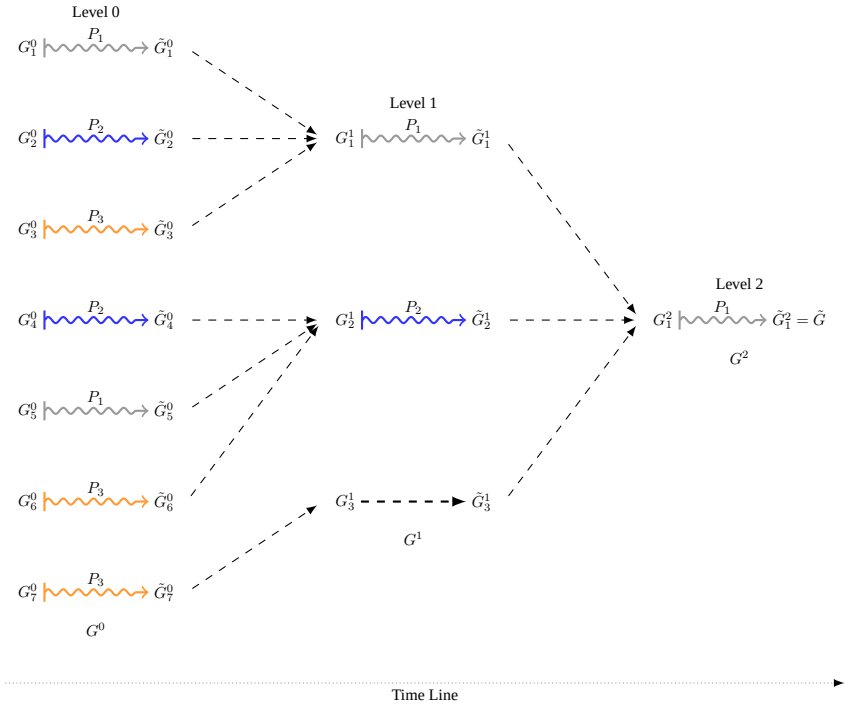


Figure 2.20: Pyramid Approach on a DAG with initially seven Atomic Subgraphs using three processes.

$$
\begin{aligned}
&X_1^1 = X_1^0; \quad Z_1^1 = Y_1^0 \cup Y_2^0; \quad Y_1^1 = Y_3^0, \\
&X_2^1 = X_4^0; \quad Z_2^1 = Y_4^0 \cup Y_5^0; \quad Y_2^1 = Y_6^0, \quad \text{and} \\
&X_3^1 = X_7^0; \quad Z_3^1 = \emptyset; \quad Y_3^1 = Y_7^0 \quad .
\end{aligned}
$$

Hence, elimination at this level yields $\tilde{G}_1^1$, $\tilde{G}_2^1$, and $\tilde{G}_3^1$ in $G^1 = G^0 - \bigcup_{i=1}^{3} Z_i^1$. In a similar manner, we get the desired bipartite graph $\tilde{G} = \tilde{G}_1^2$ at $l = 2$ as the last session by eliminating Z_1^2 vertices from G_1^2, where

$$
G_1^2 \equiv \left(\tilde{G}_1^1, \ldots, \tilde{G}_3^1\right) \quad \text{with} \quad X_1^2 = X_1^1, \quad Z_1^2 = Y_1^1 \cup Y_2^1, \quad \text{and} \quad Y_1^2 = Y_3^1 \quad .
$$

Obviously, the elimination at this level proceeds serially. In practice, it may make sense to jump prematurely to serial elimination , rather than processing until the last session, to avoid unnecessary decomposition overhead.

Lemma 2.2. *Given a DAG $G = (V, E)$ as defined by Definition (2.4) with N atomic subgraphs. Let β denote the maximum number of atomics that are to be combined together at levels $l > 0$ of pyramid algorithm shown in line 12. Furthermore, let's assume unit elimination cost of c on all subgraphs in pyramid process. Hence. the achievable speedup with P threads can be computed as*

$$
S(P) = \frac{\sum_{l=0}^{d} N_l}{\sum_{l=0}^{d} \left\lceil \frac{N_l}{P} \right\rceil} \tag{2.16}
$$

with d denoting the total number of sessions as defined by Equation (2.15).

Proof. Let $T(1)$ and $T(P)$ denote the resulting runtimes using one (serial) and P (parallel) processes, respectively. Obviously, the total number of subgraphs that are to be eliminated is given by $\sum_{l=0}^{d} N_l$ yielding a total serial time of $T(1) = \sum_{l=0}^{d} N_l \cdot c$. Moreover, execution of tasks at every session $l = 0, \ldots, d$ via P processes can be performed at a cost of $\left\lceil \frac{N_l}{P} \right\rceil \cdot c$, which results in the total parallel time of $T(P) = \sum_{l=0}^{d} \left\lceil \frac{N_l}{P} \right\rceil \cdot c$. Thus, following Amdahl's law [Amd67] we get

$$S(P) = \frac{T(1)}{T(P)} = \frac{\sum_{l=0}^{d} N_l \cdot c}{\sum_{l=0}^{d} \left\lceil \frac{N_l}{P} \right\rceil \cdot c} = \frac{\sum_{l=0}^{d} N_l}{\sum_{l=0}^{d} \left\lceil \frac{N_l}{P} \right\rceil} \quad .$$

□

Lemma 2.2 yields the speedup that can be achieved by the pyramid approach under the assumption of unity elimination cost on all considered subgraphs. Hence, for the example above with $N = 7$ and $\beta = 3$ using three processes we would expect to get a speedup of ideally

$$\frac{7 + 3 + 1}{\left\lceil \frac{7}{3} \right\rceil + \left\lceil \frac{3}{3} \right\rceil + \left\lceil \frac{1}{3} \right\rceil} = \frac{11}{5} = 2.2 \quad .$$

In the same way, we get a speedup of 2.75 for $P = 6$. Thus, duplicating the number of processes improves the speedup by a factor of 1.25. The best speedup of roughly 3.66 for this example can be achieved with $P = 7$ that guarantees enough processes to handle all tasks at every level simultaneously. However, this is not likely in practice. On the contrary, often the number of available processes is far smaller thatn the number of tasks, which might affect the speedup of this approach considerably.

2.5.3 Master-Slave Approach

The master-slave approach [BBW04] consists of two steps, namely *elimination* and *reduction*, which are illustrated for four types of atomic decompositions in Figure 2.17.

As an example we consider the atomically decomposed G in Figure 2.21 being the same as in Figure 2.20. Thereby, the vertex elimination on G_i for $i \in \{1, \ldots, 7\}$ yielding $\tilde{G}_i$ is performed by three *slave* processes P_1, P_2, and P_3 in parallel. All three processes get first three atomics and eliminate their local intermediates, whereby P_1 and P_2 are done simultaneously but earlier than P_3. This may be caused by the difference in the workload of involved processes. After termination of P_1 and P_2 the resulting local bipartite graphs $\tilde{G}_1$ and $\tilde{G}_2$ are send to the master (M), which reduces them to $\tilde{G}^1$ by eliminating $Z^1 = Y_1$ on $G^1 \equiv \left(\tilde{G}_1, \tilde{G}_2\right)$ yielding $\tilde{G}^1$. Each slave, for instance P_1, gets the next task G_5 immediately after termination of its previous job. It is worth mentioning here that the master has to check weather the eliminated subgraphs are reducible or not. For instance, the reduction of $\tilde{G}^1$, $\tilde{G}_3$, and $\tilde{G}_5$ at once is not possible since $\tilde{G}_5$ do not share any interface vertices with the first two. Thus, master reduces only $\tilde{G}_1$ and $\tilde{G}_3$ to $\tilde{G}^2$ and then, after receiving $\tilde{G}_4$, can reduce it along with $\tilde{G}_4$ and $\tilde{G}_5$ to $\tilde{G}^3$. Finally, after receiving $\tilde{G}_6$ and $\tilde{G}_7$ all subgraphs are reduced to G^4 representing the desired $\tilde{G}$. Thus, totally seven elimination and four reduction steps are performed to yield the entire bipartite graph.

2.5.4 Numerical Results

In the following we present some numerical results on PJA implemented in DALG. Here, we implement pyramid approach with $\beta = 2$ described in Section 2.5.1 using the *shared memory parallel* model with OpenMP [CDK+01] on our test system as described at the beginning of Section 2.4.2. Nonetheless, we conjecture that the master-slave to be a more suitable approach for *distributed memory parallel* models such as *message passing interface* (MPI) [Pac96, GLS99], which is the focus of ongoing implementation

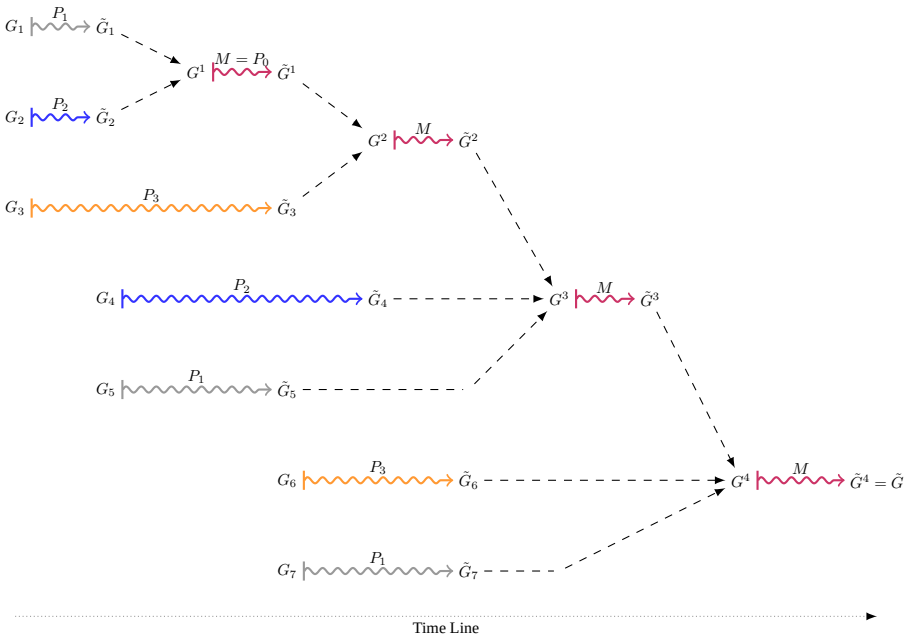


Figure 2.21: Master-Slave Approach on a DAG with initially seven Atomic Subgraphs using four processes.

activities for DALG. In addition to that, we aim to apply also *hybrid* approach [MÖ1, Qui03] using MPI and OpenMP at node interconnect and inside of a node, respectively. In this way, we hope to achieve better scalability and performance.

As test case we consider a light modification of Bratu function given in Listing 2.4. As discussed at the beginning of this section the decomposition in atomic subgraphs are supposed to be done via code instrumentation by the user as shown in lines 16–19.

Thus, the compilation of Bratu code in C++ with predefined preprocessor variable `PARALLEL_MODE` results in call of `new_atomic()` for every $i \cdot s$ iteration of the loop in line 10 with $1 \leq s \leq n-2$. The routine is supposed to mark previously generated subgraph $G_{i \cdot s}$ as atomic after executing the $i \cdot s$-th loop for $i \in \{1, \ldots, n-2\}$. Hence, totally $\left\lceil \frac{n-2}{s} \right\rceil$ atomic subgraphs are generated by calling `bratu` for a particular n. Hence, we get $n-2$ subgraphs for $s = 1$, which is the case for tests performed below. Obviously, the size of local subgraphs grows with s, whereas theirs total number decreases. Hence, s can be used to change the workload of concurrent processes as shown in Figure 2.22 (f). Moreover, we manage to combine two neighboring eliminated subgraphs ($\beta = 2$ in Listing 2.3) before proceeding with the next parallel session.

Listing 2.4: Instrumented Bratu

```
1 void bratu(int n, double** x, double l, int s) {
2   double h = 1./(n-1);
3   double r[n][n];
4   // enforce boundary condition
5   for (int i = 0; i < n; i++) {
6     x[i][0] = 0.;    x[i][n-1] = 0.;    x[0][i] = 0.;
7   }
```

```
8   for (int i = 0; i < n; i++)  x[n-1][i] = 1.;
9   // iterate over inner points
10  for (int i = 1; i < (n-1); i++) {
11    for (int j = 1; j < (n-1); j++) {
12      r[i][j] = 0. - ((x[i+1][j] - 2 * x[i][j] + x[i-1][j]) / (h*h))
13      - ((x[i][j+1] - 2 * x[i][j] + x[i][j-1]) / (h*h))
14      - l * exp(x[i][j]);
15    }
16      #if (PARALLEL_MODE)
17      if (i%s == 0)
18        new_atomic();
19      #endif
20  }
21  // updating the inner points
22  for (int i = 1; i < n-1; i++)
23    for (int j = 1; j < n-1; j++)
24      x[i][j] = r[i][j];
25 }
```

Let us now consider Figure 2.22 (a) resp. (c) representing runtime results of PJA of Bratu function by DALG in forward mode on DEJ resp. CRS using one (LFM #1) and eight (LFM #8) threads, which we compare with the corresponding global forward elimination ordering (GFM). Analog, proceeds in (b) and (d) for reverse elimination ordering denoted by LRM and GRM, respectively. On both DEJ and CRS, the runtime gainings achieved by local elimination (on atomics) by a single thread is in order of magnitude better than the global one. Therefore, considering Table 2.3 LFM#1 [LRM#1] on DEJ is about a factor of $15.5 = \frac{3162}{204}$ [$5,3 = \frac{1082}{203}$] faster than GFM [GRM], which could be surprising at first glance. Thus, the achieved high runtime gainings does not seem to be the benefit of parallelization. We conjecture it to be rather caused by the much smaller search space for dependencies needed to eliminate a particular row on atomic blocks than on the entire DEJ. Thereby, the elimination of p intermediates can be performed at computation cost of $O(p \cdot q^2)$. A single row $i \in Z$ can be eliminated at a cost of $(i-1)\cdot(q-i) \leq q^2$, whereas in case of t atomic decompositions the same intermediate can be eliminated at a cost of $(i-1) \cdot \frac{q-i}{t} \leq \frac{q^2}{t}$ yielding a total cost of $O(p \cdot \frac{q^2}{t})$. Hence, we may gain theoretically a factor of $t = 98$ for $n = 100$ in searching after dependencies just by decomposition of the underlying matrices. Moreover, the gainings are more substantial on CRS in both forward and reverse modes that we believe to be a consequence of sparsity exploitation. Thereby, even thought $1.7 \approx \frac{318290}{185402}$ times less multiplications are performed in forward mode than in reverse mode the latter is roughly $2.1 \approx \frac{30}{14}$ times faster than the former, which we attribute to the better performance of linear search by an average factor of roughly $3.8 \approx \frac{0.321536}{0.0843}$. Hence, the real achieved speedup using eight threads in forward [reverse] mode on DEJ

Elimination Mode	Time(DEJ)	Time(CRS)	#Muls	#Fill-in	#Entries	ω
GFM	3162 sec.	7159 sec.	362600	343392	544684	0.531108
GRM	1082 sec.	1556 sec.	191688	172480	373772	0.0657927
LFM (#1)	204 sec.	30 sec.	185402	166194	367486	0.321536
LFM (#8)	121 sec.	17 sec.	185402	166194	367486	0.321536
LRM (#1)	203 sec.	14 sec.	318290	299082	500374	0.0843
LRM (#8)	118 sec.	9 sec.	318290	299082	500374	0.0843

Table 2.3: Summary of Measurement Data for Bratu in Parallel Mode for $n = 100$.

and CRS is roughly $1.6 \approx \frac{204}{121}$ [$1.7 \approx \frac{203}{118}$] and $1.7 \approx \frac{30}{17}$ [$1.5 \approx \frac{14}{9}$] as shown in Figure 2.22 (e). The optimal speedup from Equation (2.16) with eight threads in the pyramid model for $n = 100$ yielding

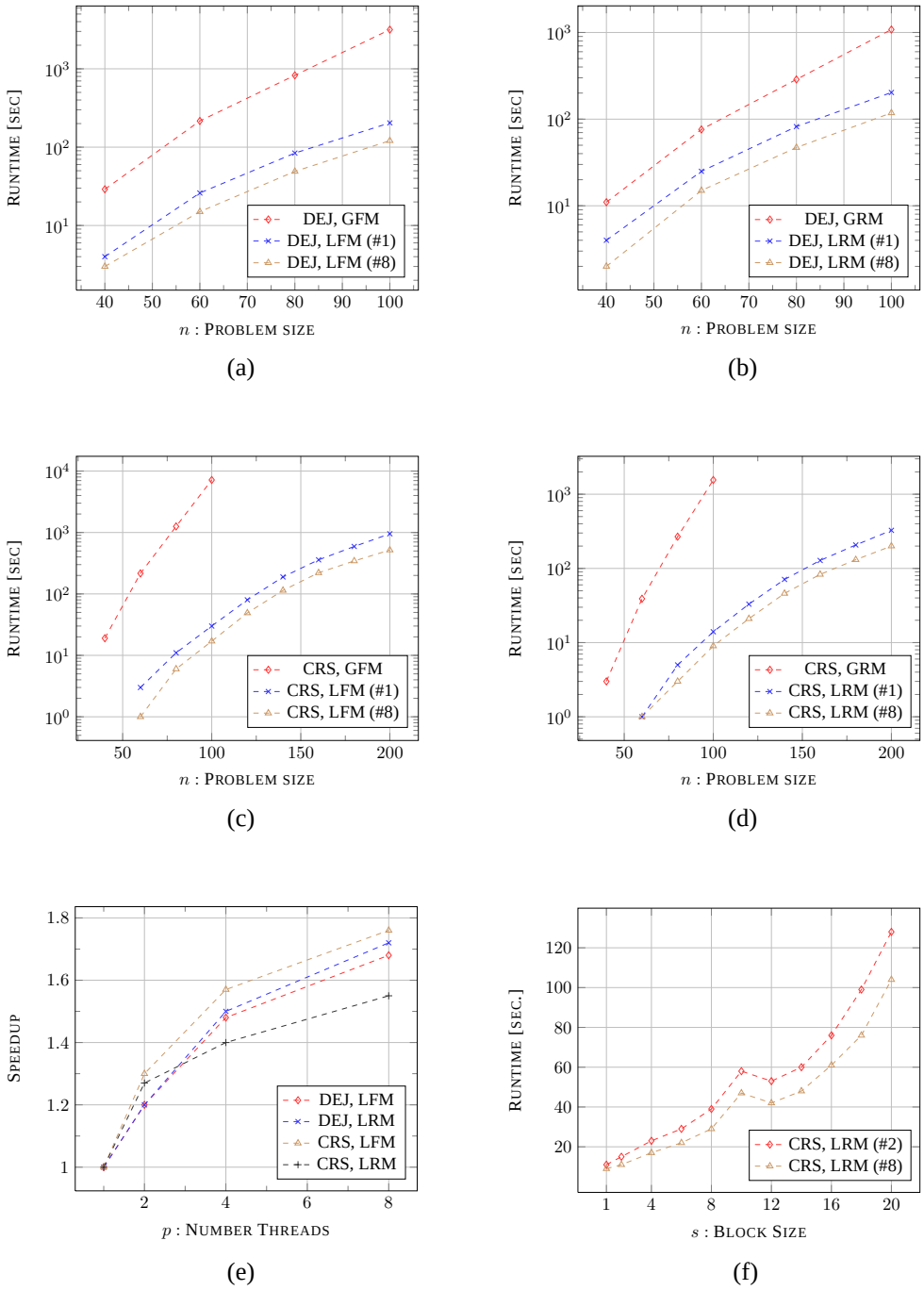


Figure 2.22: Runtime Results on Parallel Jacobian Accumulation by DALG on Bratu.

$n-2=98$ atomics for $\beta=2$ would be

$$\frac{98+49+25+13+7+4+2+1}{\lceil\frac{98}{8}\rceil+\lceil\frac{49}{8}\rceil+\lceil\frac{25}{8}\rceil+\lceil\frac{13}{8}\rceil+\lceil\frac{7}{8}\rceil+\lceil\frac{4}{8}\rceil+\lceil\frac{2}{8}\rceil+\lceil\frac{1}{8}\rceil}=\frac{199}{29}\approx 6.86 \quad .$$

Thus, we observe that the parallelization itself does not seem to perform well as desired. Moreover, a closer look to the line 12 of the Bratu function shows that a `r[i][j]` contributes directly to an output. The latter may depend on common independents, but not on each other. Moreover, no independent is overwritten in the for loop. This case is illustrated in CASE 1 of Figure 2.17. Hence, the reduction step is nothing else than absorbing edge labels with no vertex elimination because of empty interfaces.

To conclude the discussion on PJA, we illustrate in Figure 2.22 (f) the impact of workload on runtime of parallel mode for $n=100$ and $s=1,2,4,6,8,10$ with s denoting the block sizes. The respective code fragment is shown in line 17 of Listing 2.4. We observe that increasing the size (s) of atomic blocks slows down the parallelization considerably on both DEJ and CRS, respectively. We believe that the reason for this lies again in the increasing size of search spaces on submatrices, which grow as s increases. Nonetheless, we observed so far that PJA has the potential to accelerate the Jacobian accumulation on both DEJ and its CRS counterpart considerably. As suspected at the end of Section 2.4.2 the main contribution to the speedup seems to be a side effect of smaller search spaces for dependencies within (atomic) sub-matrices. However, we did not observe the theoretical factor of roughly seven as defined by Equation (2.16) with eight threads for our test case. Hence, further research is planned to be invested on improving the performance of PJA using Pyramid approach.

2.6 Iterative Jacobian Accumulation

Sparsity exploitation of extended Jacobians using the corresponding compressed row representations tends to decrease the memory consumption as our experimental results previously have shown. However, our assumption so far was that there is enough memory to store the entire extended Jacobian or its bit pattern/CRS of the underlying function. Thus, we are still in the situation, where the memory bounds the capability of Jacobian accumulation, which is the main common problem of any AD approach that aims to accumulate derivatives on any kind of internal representation. In the following we present iterative Jacobian accumulation to deal with this problem.

Here, we use the DAG representation instead of the extended Jacobian that are conceptually equivalent as discussed at the beginning of this chapter. For illustration, we consider G in Figure 2.23 to represent the DAG of our example function for $n=2$. Its independent and dependent vertices are given by $1,2$ and $9,10$, respectively. Hence, elimination of the intermediate vertices $3,4,5,6,7,8$ yields the complete bipartite graph $\tilde{G}=K_{2\times 2}$ with $1,2$ as source and $9,10$ as target vertices. Edge labels are missing in the following examples just for simplicity Let us assume now that only the subgraph G_7 as shown in Figure 2.24 with vertices $1,\ldots,7$ along with their incoming edges fits into the available memory. Now, we eliminate vertices from G_7 to free memory. Therefore, we need **local information about eliminatable vertices** of G_7. As mentioned above the vertices 1 and 2 are independent and hence not eliminatable. We can not eliminate 7 either because it is not locally detectable if it is used later or not. A DAG vertex is used if it appears as a predecessor of some other vertices. The same argumentation holds in fact for the vertices 4, 5, and 6, which are known to be alive as they represent the current instances of t, x_1, and x_2 of F, respectively. Consequently, they may be multiply used during the evaluation process of F, such that we have to mark them also as not eliminatable. Thus, we get 3 as the only eliminatable vertex. Its elimination yields $\tilde{G}_7$ by the generation

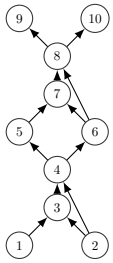


Figure 2.23: DAG of F defined by Equation 1.18 for $n=2$.

of the new edge $(1,4)$ followed by deleting vertex 3 and its incident edges $(1,3),(2,3)$, and $(3,4)$. Hence, we gain memory for two edges, which enables us to add the vertex 8 with its incoming edges $(7,8)$ and $(6,8)$ into $\tilde{G}_7$ yielding G_8. We know also that vertex 4 and 8 correspond to the same program variable t with 8 representing its current instance, which implies that 4 is not alive anymore and hence can be eliminated. After the generation of vertex 8 and after single use of vertex 7 we mark it as eliminatable in G_8 as it corresponds to a temporary (not program) variable in F. Now, we eliminate 4 and 7 and yield $\tilde{G}_8$, where we gain memory for two edges and hence can build G_{10} by adding the last two vertices 9 and 10 along with their incoming edges $(8,9)$ and $(8,10)$ to $\tilde{G}_8$. Finally, we get $\tilde{G} = \tilde{G}_{10}$ by eliminating 5, 6, and 8.

One aspect of the iterative approach that should be pointed out here is that regardless of the local elimination ordering of the choice is, the resulting global ordering might be different. We refer here to the resulting ordering as *cross-country* ordering as proposed by Griewank [GW08]. Moreover and obviously, the resulting fill-in pattern can also be different. Therefore, consider Figure 2.24, where we apply forward elimination ordering locally. The resulting ordering is $3,4,7,5,6,8$ differs from the global one as $3,4,5,6,7,8$. Thereby, the resulting fill-ins are ten and fifteen in the former and latter, respectively. In

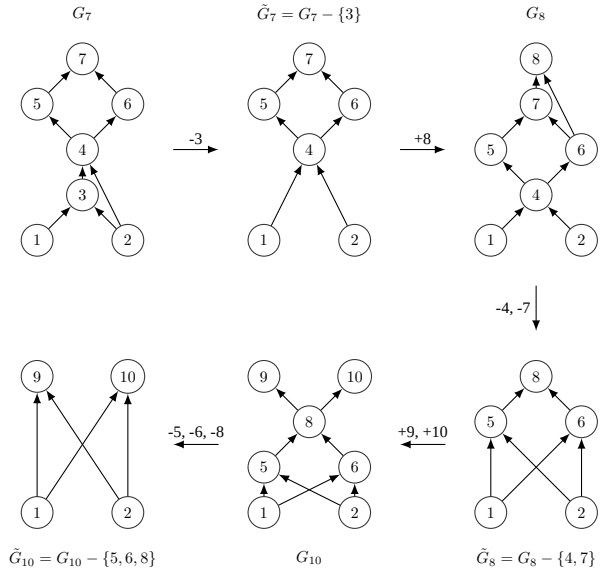


Figure 2.24: Iterative Vertex Elimination. The „-" Prefix to a Vertex Index means that it is eliminated, whereas „+" indicates its Generation.

order to illustrate the idea behind the iterative Jacobian accumulation by vertex elimination, we consider G of F to be the result of the statement level execution of the respective SAC of Equation (1.2) as

$$G_j = (V_j, E_j) \quad \text{with} \quad V_j = V_{j-1} \cup \{j\} \quad \text{and} \quad E_j = E_{j-1} \cup \{(i,j) : i \prec j\} \tag{2.17}$$

for $j = 1,\ldots,q$ with $G = G_q$ and $G_0 = \emptyset$. Furthermore, the independent X_j, dependent Y_j, and intermediate Z_j vertices of G_j are defined as

$$X_j = X, \quad Y_j = (D_j - X_j) \cup \{i \in V_j : S_i = \emptyset\}, \quad \text{and} \quad Z_j = V_j - (X_j \cup Y_j) \quad .$$

Here, $D_j = \{i \in V_j : v_i \text{ is alive SAC variable}\}$ denotes the set of all alive vertices of G_j. Clearly, $G_i = (\{1,\ldots,i\},\emptyset)$ for independent vertices i.

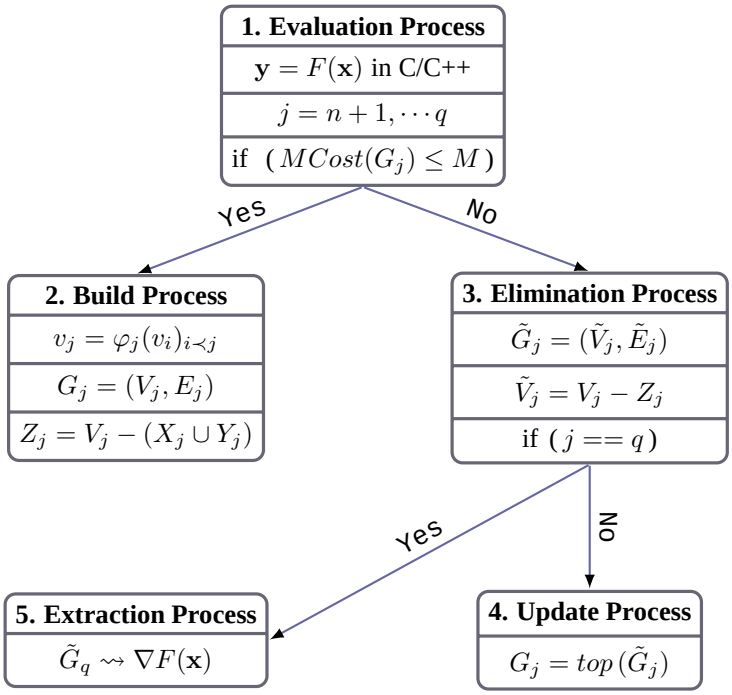


Figure 2.25: Iterative Process of Jacobian Accumulation by Vertex Elimination.

Figure 2.25 illustrates IJA on a restricted memory M, where every box represents a process. Arrows represent the transitions between processes, where each of them is labeled with the result of the condition at the end of the corresponding source process. For instance, `Yes` on the arrow from process 1 to process 2 means that the condition $Mem(G_j) \leq M$ at the end of the evaluation process is satisfied and hence G_j can be built on the existing memory M, where

$$MCos(G_j) = Mem(G_{j-1}) + |E_j| \cdot \mu_e$$

denotes the memory consumption of G_j that follows immediately from Equation (1.19). Furthermore, we assume that the termination of the leaf processes 2, 4, and 5 are followed by a jump to the root process 1. The latter jump happens if the Jacobian at another point is of interest. Otherwise, process 5 is supposed to finalize IJA process. Here, the evaluation process initiates the generation (process 2) of G_j based on G_{j-1} for $1 \leq j \leq q$ as long as G_j fits into M. Otherwise, it starts the vertex elimination process yielding the eliminated DAG

$$\tilde{G}_j = (\tilde{V}_j, \tilde{E}_j) \quad \text{with} \quad \tilde{V}_j = V_j - Z_j \quad \text{and} \quad \tilde{E}_j = E_j - \{(i,k) \mid i \in Z_j \vee k \in Z_j\} \quad .$$

In case of $j == q$ at the end of the elimination process we get $\tilde{G} = \tilde{G}_q$ representing the bipartite graph of G defined by Equation (1.15). Hence, the nonzero entries of $\nabla F(\mathbf{x})$ can be obtained (process 5) from $\tilde{G}$ just by reading its edge labels. Otherwise, and before we proceed with the evaluation again, we set $G_j = top\,(\tilde{G}_j)$ representing the topological reordered version of $\tilde{G}_j$, so that $V_j = top\,(\tilde{V}_j)$ according to Equation (2.6).

Theorem 2.2. *Given the linearized DAG G of F as defined by Equation (*1.13*) with the intermediate vertices Z. The resulting elimination graphs $G - \sigma_1(W)$ and $G - \sigma_2(W)$ after eliminating the vertices*

$W \subseteq Z$ in two different orders σ_1 resp. σ_2 are equal, that is,

$$\forall \sigma_1 \neq \sigma_2 : G - \sigma_1(W) = G - \sigma_2(W) \quad .$$

Proof. Mosenkis shows this in section 3 of [MN10] about minimum edge count problem on linearized DAGs. □

From Theorem 2.2 it follows immediately that the memory cost of the eliminated DAG $G - W$ does not depend on the ordering in which the vertices W are eliminated, that is ,

$$\forall \sigma_1 \neq \sigma_2 : Mem(G - \sigma_1(W)) = Mem(G - \sigma_2(W)) \quad .$$

Furthermore, let $MGain(G - i) = Fillout(G - i) - Fillin(G - i)$ denote the memory savings in the number of edges after the elimination of vertex $i \in W$ from G with $Fillout(G - i)$ and $Fillin(G - i)$ denoting the respective number of fill-out and fill-in edges, respectively. It follows easily that

$$MGain(G - W) = |E(G)| - |E(G - W)| = \sum_{i \in W} MGain(G - i) \quad ,$$

where $MGain(G - W)$ denotes the total memory savings as a sum over the local memory savings $MGain(G - i)$ by eliminating all vertices $i \in W$. Even though the total memory savings for all possible orderings of vertices W are the same, the local savings in general do depend on the elimination ordering and hence might be different.

An example is given in Figure 2.26, where the elimination of vertex 3 and vertex 4 in two possible ordering $3, 4$ and $4, 3$ result in the same eliminated graph $G - \{3, 4\}$. However, eliminating vertex 3 first yields $G_7 - \{3\}$ with five edges, thereby reducing the size of G by two edges. Further elimination of vertex 4 results in the bipartite graph $G_7 - \{3, 4\}$ with six edges, which means a memory loss of one edge comparing with $G_7 - \{3\}$. On the contrary, eliminating first vertex 4 results in $G - \{4\}$ with eight edges, meaning memory loss of one edge. Additional elimination of vertex 3 saves two edges, yielding a total saving of one edge. Note that the total saving of one edge is the same for both orderings.

In fact, this addresses one problem of the iterative approach, where at certain iteration points the local elimination (see process 3 of Figure 2.25) may exceed the available memory bound M by adding more fill-in edges than freeing fill-out ones. In order to illustrate this, let us consider again Figure 2.26 with $M = 7$ as the available memory. Eliminating first vertex 3 yields $G_7 - \{3\}$, which still fits into the memory, whereas eliminating vertex 4 does not because $G_7 - \{4\}$ needs eight edges. Thus, at every elimination step the general combinatorial problem is to keep the DAG within the memory bound, which is formulated as follows.

Problem 2.1. *Given the DAG $G = (V, E)$ as defined by Equation (*1.13*) with $Z \subseteq V$ denoting the eliminatable vertices, find a subset $W \subseteq Z$ of eliminatable vertices Z and an appropriate elimination ordering σ that satisfies the memory bound over the entire elimination of vertices W, that is,*

$$\sum_{i=1}^{|W|} Mem(G - \sigma(W_i)) \leq Mem(G) - M \quad \forall\, W_i \subseteq W \quad \textit{with} \quad |W_i| = i \quad .$$

On the other hand, even though if we found an appropriate subset of eliminatable vertices along with an appropriate ordering that would solve Problem (2.1), this alone does not guarantee the success of the iterative approach in general. This becomes clear when we consider the building process (2) in Figure 2.25 that generates eliminatable vertices, such that $Z_j \neq \emptyset$. In worst case it could be the case that Z_j is empty or, as discussed previously, that the elimination process does not free a sufficient amount of memory needed to proceed further with building DAG.

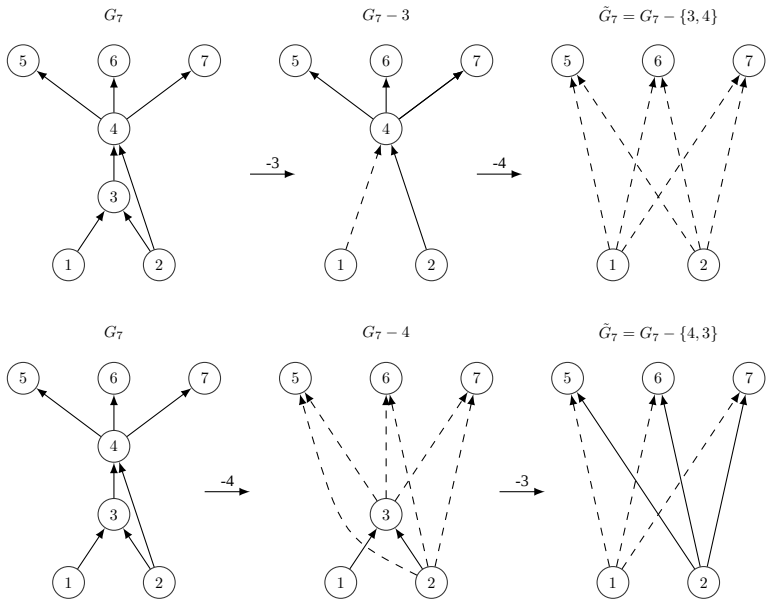


Figure 2.26: Memory Issues on Elimination Process of IJA, where dashed Edges mark generated ones.

To tackle this problem, let us consider G_7 in Figure 2.27 with the memory bound $M = 7$. Furthermore, let us assume the next build process would add the vertex 8 with its incoming edges $(5, 8)$ and $(7, 8)$ to $\tilde{G}_7$ yielding G_8 representing the final graph with $Z_8 = \{4, 5, 7\}$. In case of eliminating vertex 3 and 4 in a sequence we would end up with $G_7 - \{3, 4\}$ with six edges that frees storage only for one edge, whereas we need two of them for G_8. One possible solution could be to eliminate first vertex 3, freeing two edges, and then add vertex 8 with its incoming edges to $\tilde{G}_7$. Now, we eliminate vertices 5, 7, and 4 consecutively to satisfy the memory limit of 7 edges yielding the bipartite graph $\tilde{G}_8 = G_8 - \{5, 7, 4\}$ with totally six edges.

So far we have discussed the combinatorics involved in IJA. Thereby, a locally conservative information about eliminatable vertices is considered to be safe, also referred to as *elimination safe*, to guarantee the correctness of the elimination process. Furthermore, not every elimination ordering seems to satisfy the memory boundary condition as formulated in Problem (2.1); some orderings may tend to exceed it during the local elimination. In this case IJA fails to accumulate the Jacobian of F. One way to deal with this problem is to allocate memory space for local DAGs G_j in Equation (2.17) at every evaluation step conservatively. Therefore, we consider the complete DAG

$$\widehat{G}_j = (\widehat{V}_j, \widehat{E}_j) \quad \text{with} \quad \widehat{V}_j = V_j \quad \text{and} \quad \widehat{E}_j = \{(i,k) \mid \forall\, i,k \in \widehat{V}_j \,:\, i < k\} \supseteq E_j \tag{2.18}$$

of G_j, where every vertex $i \in \widehat{V}_j$ has incoming edges from all previous vertices, hence $|P_i| = i - 1$. Furthermore, $\widehat{Z}_j$ denotes the intermediate vertices of $\widehat{G}_j$. Thus, the memory cost of $\widehat{G}_j$ represents an upper bound for the memory cost of G_j as formulated in the following lemma.

Lemma 2.3. *Let $G = (V, E)$ denote a DAG as defined by Equation (*1.13*) and let $\widehat{G} = (\widehat{V}, \widehat{E})$ denote its complete version as defined in Equation (*2.18*).*

$$Mem(G) \leq Mem(\widehat{G}_j) := \sum_{i=1}^{q} (i-1) \cdot \mu_e \tag{2.19}$$

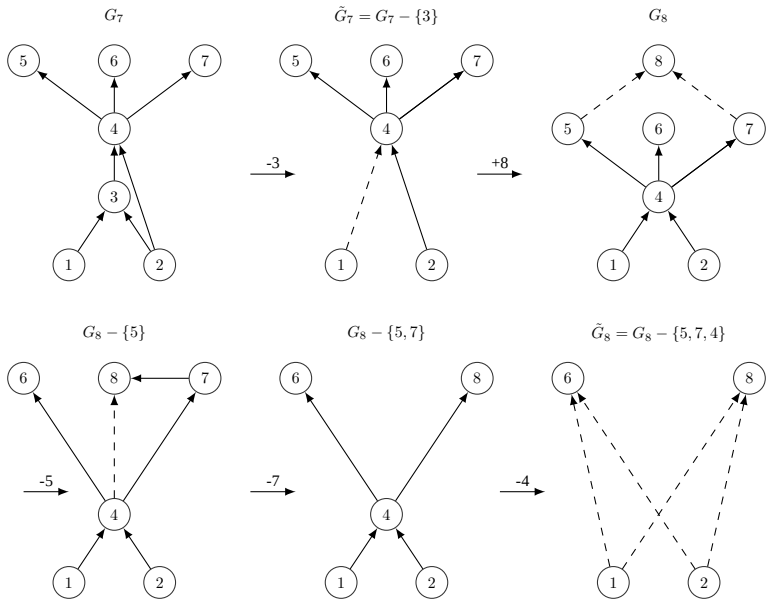


Figure 2.27: Memory Issues on Build Process of IJA.

Proof. The proof follows immediately from Equation (1.19) and the fact that a vertex $i \in V$ can have at most $i-1$ predecessors, that is,$|P_i| \leq i-1$. □

Furthermore, the following lemma shows that the elimination of an intermediate vertex $i \in \widehat{Z}_j$ yields again a complete DAG $\widehat{G}_j - i$ with no fill-in edges. This means that, eliminating a particular vertex on a complete DAG results in another complete one.

Lemma 2.4. *Let $\widehat{G} = (\widehat{V}, \widehat{E})$ be the complete DAG of G defined by Equation (2.18) with $j \in \widehat{V}$ denoting an intermediate vertex. Furthermore, we assume that $\widehat{G}$ is also topologically ordered with respect to their dependencies, that is, an edge $(i,k) \in \widehat{E}$ implies that $i < k$. Thus, $Fillin(\widehat{G} - j) = \emptyset$ and hence $\widehat{G} - j$ is complete.*

Proof. First we show in (1) by contradiction that eliminating j produces no fill-in, which we refer to as *No-Fillin property*. Then we show in (2) that $\widehat{G} - j$ is complete that we refer to as *Completeness property*.

1. **No-Fillin property**: By contradiction we show that $Fillin(\widehat{G} - j) = \emptyset$. Therefore, let us assume that during the elimination of j we generate a new edge $(i,k) \notin \widehat{E}$. Thus, we have $i \in P_j$ and $k \in S_j$ because (i,k) is generated by eliminating j. Thus, from the topology of $\widehat{V}$ it follows that $i < j$ and $j < k$, hence $i < k$. However, this would mean that $\widehat{G}$ is not complete, which then contradicts the definition of $\widehat{G}$.

2. **Completeness property**: Because of the topological ordering of the vertices of $\widehat{G}$ we have

$$\forall i,k : i \in P_j \quad \text{and} \quad k \in S_j \quad \Rightarrow \quad i < k \quad .$$

   This means that there are direct edges from every predecessor of j to all of its successors in $\widehat{G}$. Hence, $\widehat{G} - j$ is also complete.

□

Obviously, Lemma 2.4 holds also for the elimination of a subset W of intermediates $\widehat{Z}_j$. More substantially, the elimination ordering affects neither the Completeness nor the No-Fillin property and hence can be ignored in this context. Consequently, it becomes clear that the elimination of the intermediate vertices $\widehat{Z}_j \subset \widehat{V}_j$ in arbitrary order can not exceed the given memory bound, which solves exactly the problem during the local elimination as formulated in Problem (2.1). Thus, conservative memory accumulation for G_j prevents us from running out of memory during the elimination process. We note here that this still does not guarantee the success of IJA as discussed below in Figure 2.28. However, an upper bound for the number of vertices of G_j can be determined for a given memory bound M as formulated by the following lemma.

Lemma 2.5. *Let M represent the available memory in bits. The DAG $G = (V, E)$ as defined by Equation (*1.13*) can have at most*

$$q = \frac{1 + \sqrt{1 + \frac{8 \cdot M}{\mu_e}}}{2} \tag{2.20}$$

vertices.

Proof. From Equation (2.19) it follows that

$$Mem(G) \leq Mem(\widehat{G}) = \sum_{j=1}^{q} (j-1) \cdot \mu_e = \frac{q \cdot (q-1)}{2} \cdot \mu_e \quad .$$

Hence, for the given memory bound M we get

$$\frac{q \cdot (q-1)}{2} \cdot \mu_e = M \Leftrightarrow q^2 - q - \frac{2 \cdot M}{\mu_e} = 0 \quad .$$

The binomial formula yields the following two solutions of the resulting polynomial above

$$q = \frac{1 \pm \sqrt{1 + \frac{8 \cdot M}{\mu_e}}}{2} \quad ,$$

where $1 + \frac{8 \cdot M}{\mu_e}$ is a positive number ≥ 1 and square root and division operators are supposed to return integer values. □

To support the discussion above let us consider again G_7 in Figure 2.27, where a memory bound of seven edges ($M = 7$) was taken to accumulate the corresponding bipartite graph $\tilde{G}_8$ iteratively. Our focus in the following is on the conservative memory allocation and its impact on the entire iterative model shown illustrated in Figure 2.28. Therefore, we replace the memory condition $Mem(G_j) \leq M$ in the evaluation process of Figure 2.25 by $Mem(\widehat{G}_j) \leq M$ with

$$MCos(\widehat{G}_j) = Mem(\widehat{G}_{j-1}) + |\widehat{V}_{j-1}| \cdot \mu_e \quad .$$

From Equation (2.20) it follows for $M = 7 \cdot \mu_e$ bits that conservatively four vertices would fit into the available memory as

$$q = \frac{1 + \sqrt{1 + 8 \cdot 7}}{2} = 4 \quad .$$

Thus, the evaluation process initiates the elimination once G_4 with $Mem(\widehat{G}_4) = 1 + 2 + 3 = 6$ is constructed. The reason is that inserting vertex 5 with an additional memory requirement of conservatively four edges would exceed the memory bound of seven edges. However, the elimination of vertex 3 yields

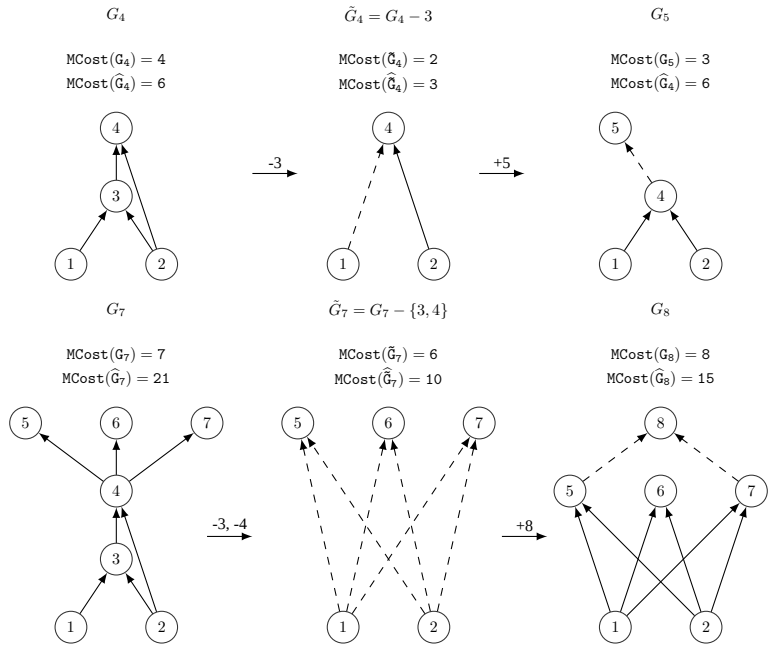


Figure 2.28: Conservative Memory Approach.

$\tilde{G}_4$ with real two and conservatively three edges. Now, inserting vertex 5 into $\tilde{G}_4$ yields G_5, while consuming additionally three edges and yielding $\widehat{\tilde{G}}_5 = 6$. Unfortunately, no elimination is possible anymore because vertex 4 is known to be alive, so that the iterative approach fails to compute $\tilde{G}_8$.

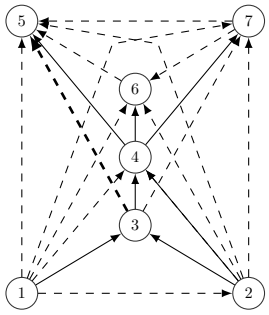
Let us now try to put $\widehat{G}_7$ entirely into memory. Therefore, we would need a memory for $\frac{7 \cdot (7-1)}{2} = 21$ edges. Taking now $M = 21$ as memory bound would cause the iterative approach to succeed as illustrated in the bottom row of Figure 2.28. Thereby, the elimination of 5 and 7 from G_8 yields the same bipartite graph as $G_8 - \{5, 7, 4\}$ in Figure 2.27.

At this point we recapitulate that taking the memory consumption of the complete variant of a given DAG is considered the worst case solution of Problem (2.1), where we try to avoid running out of memory during the elimination process. Here, all those edges not in the original DAG are considered a „place holder" for potential fill-in. Obviously, the memory consumption of the complete DAG is equivalent to that of the extended Jacobian in dense format i.e. DEJ. Each entry of the latter is represented by an edges of the former as discussed in the following.

2.6.1 Iterative Approach on Extended Jacobians

The relation between the linearized DAG G and the extended Jacobian C' of F was the focus of the discussions at the beginning of this chapter. An example was given in Figure 2.3, where the nonzero sub-diagonal entries of C' correspond to the edges of G. Moreover, the complete DAG $\widehat{G}$ relates even more to C' as all those edges of $\widehat{G}$ not contained in G represent exactly the zero sub-diagonal entries of C'. An example is given in Figure 2.29, where dashed edges in $\widehat{G}$ correspond to zeros of C'. Thus, it is not surprising that the memory cost of $\widehat{G}$ is in the same complexity class of C'. This becomes clear when we consider the memory cost of the former and latter in Equation (2.19) and Equation (2.7) and the resulting number of allocatable vertices and rows as suggested in Lemma 2.5 and Lemma 2.6, respectively. However, the main difference is made by required storage for edges μ_e and floating values μ_F, where we

consider $\mu_e \leq \mu_F$ reasonable as edges are labeled with floating point values of the respective local partial derivatives.



$\widehat{G}$

$$C' = \begin{pmatrix} 1 & & & & & & \\ 0 & 2 & & & & & \\ 1 & 1 & 3 & & & & \\ 0 & 1 & 1 & 4 & & & \\ 0 & 0 & 0 & 1 & 5 & & \\ 0 & 0 & 0 & 1 & 0 & 6 & \\ 0 & 0 & 0 & 1 & 0 & 0 & 7 \end{pmatrix}$$

Figure 2.29: The complete DAG and the respective Extended Jacobian.

Lemma 2.6. *Let M represent the available memory in bits. The extended Jacobian C' can have at most*

$$q = \frac{1 + \sqrt{1 + \frac{8 \cdot M}{\mu_F}}}{2} \tag{2.21}$$

rows.

Proof. From Equation (2.7) it follows that

$$M = \sum_{j=1}^{q} (j-1) \cdot \mu_F = \frac{q \cdot (q-1)}{2} \cdot \mu_F \Leftrightarrow \frac{2 \cdot M}{\mu_F} = q^2 - q \Leftrightarrow q^2 - q - \frac{2 \cdot M}{\mu_F} = 0 \quad .$$

The binomial formula yields the following two solutions of the resulting polynomial above

$$q = \frac{1 \pm \sqrt{1 + \frac{8 \cdot M}{\mu_F}}}{2} \quad ,$$

where $1 + \frac{8 \cdot M}{\mu_F}$ is a positive number ≥ 1. □

In the following we consider the update process on C', which is, in graphical term, nothing else than topological reordering of DAG vertices. However, on the extended Jacobian the topological reordering of rows can mean copying nonzero elements from old locations into new ones. It can easily be shown that the topological ordering guarantees that there is enough memory in the new location of a row. The proof idea can be described as follows. Therefore, let us assume that row k of the extended Jacobian C' as shown in Figure 2.30 is free to be reused after its elimination and that there is no free rows before it. Furthermore, let us assume that only rows i, j with $k+1 < i < j$ are not eliminated so far, such that the set of not eliminated rows can be denoted by $D = \{1, \ldots, k-1, i, j\}$.

Hence, i and j are potential candidates to be moved into row k after its elimination yielding $C' - k$. It is obvious that row j can not be moved to k as it depends on $i > k$ and hence does not fit into k. Moreover, moving j into k would also violate the topological ordering as $i < j$. However, row i fits into location of k for sure as there is no row $l \in \{k+1, \ldots, i-1\}$ with $l \prec i$ by assumption. Otherwise, row i would be faced with the same problem as row j did before. Thus, a possible topological ordering

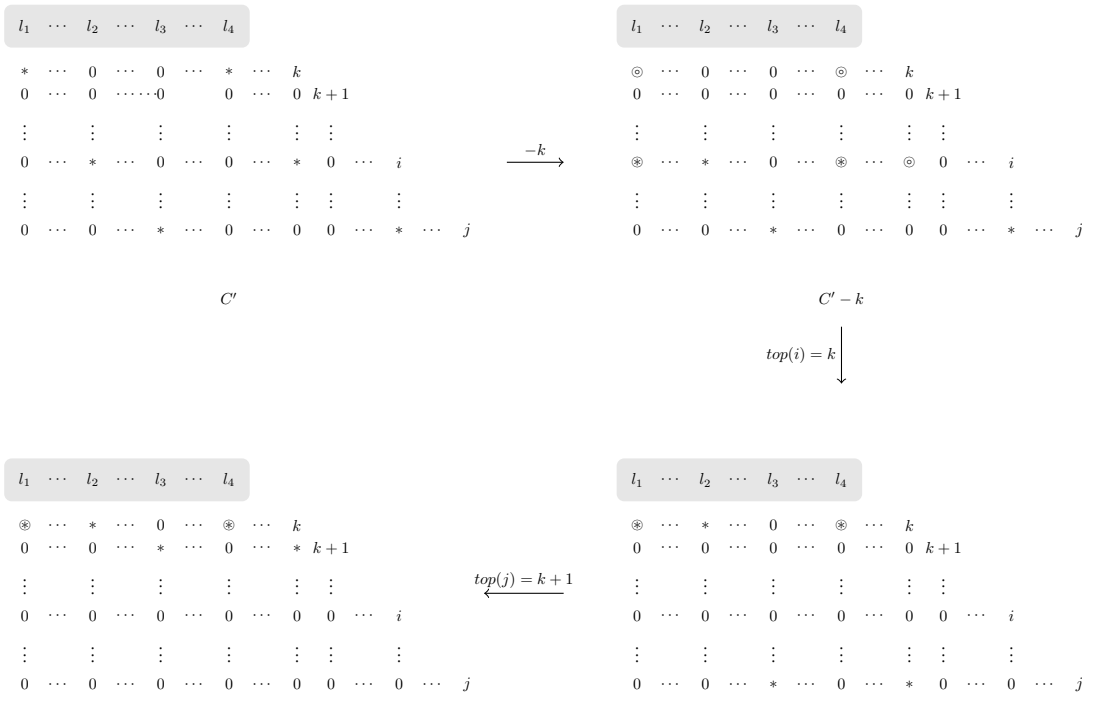


Figure 2.30: Update Step on Extended Jacobians.

as defined by Equation (2.6) with $Z = D$ is as

$$top\,(1) = 1 \quad \ldots \quad top\,(k-1) = k-1 \quad top\,(i) = k \quad top\,(j) = k+1 \quad .$$

This means that we keep rows $1, \ldots, k-1$ unchanged, whereas row i resp. j are moved into k resp. $k+1$. Thus, all rows $k+2, \ldots, j$ are freed and thus can be used again during the evaluation process. Note that the dependency of row j on i has also to be maintained accordingly, that is, after moving row i into k row j depends on row k. The following example illustrates the iterative process on the extended Jacobian of our example function for $n = 2$. The corresponding DAG version has already been illustrated in Figure 2.24 with a vertex [edge] number of totally seven [twenty one].

Example 2.7. *For better illustration we consider a light modification of the SAC of our example function as shown in Example 1.1 as follows.*

$$\begin{aligned}
&\textit{for } i = 1, \ldots, n \\
&\quad v_1^i = x_1; \quad v_2^i = x_2; \\
&\quad v_3^i = v_1^i \cdot v_2^i; \\
&\quad v_4^i = v_3^i - v_2^i; \\
&\quad t = v_4^i; \\
&\quad v_5^i = \sin(v_4^i); \\
&\quad v_6^i = \exp(v_4^i); \\
&\quad x_1 = v_5^i; \quad x_2 = v_6^i;
\end{aligned}$$

Here, v_j^i denotes the ith instance of the SAC variable v_j. Furthermore, we replace the diagonal entries by the corresponding SAC variables and denote an eliminated row by the diagonal entry $v_\cdot$. Analogous to the DAG version, we assume a memory limit of seven (twenty one sub-diagonal entries) extended Jacobian rows. Thus, the evaluation process yields the extended Jacobian A_1 with nonzero entries denoted by c_i for $i = 1, \ldots, 8$. For instance, $c_3 = 1$ represents the local partial derivative of v_4^1 with respect to v_3^1. Now, eliminating row v_3^1 yields A_2 with $c_3 = c_3 + c_2 \cdot c_4$ and $c_9 = c_1 \cdot c_4$ as follows.

$$
\begin{array}{ccccccc}
v_1^1 & & & & & & \\
0 & v_2^1 & & & & & \\
c_1 & c_2 & v_3^1 & & & & \\
0 & c_3 & c_4 & v_4^1 & & & \\
0 & 0 & 0 & c_5 & v_5^1 & & \\
0 & 0 & 0 & c_6 & 0 & v_6^1 & \\
0 & 0 & 0 & 0 & c_7 & c_8 & v_3^2
\end{array}
\qquad A_1
$$

$$
\begin{array}{ccccccc}
v_1^1 & & & & & & \\
0 & v_2^1 & & & & & \\
\circledcirc & \circledcirc & v_\cdot & & & & \\
c_9 & c_3 & \circledcirc & v_4^1 & & & \\
0 & 0 & 0 & c_5 & v_5^1 & & \\
0 & 0 & 0 & c_6 & 0 & v_6^1 & \\
0 & 0 & 0 & 0 & c_7 & c_8 & v_3^2
\end{array}
\qquad A_2
$$

The update process copies rows $i = 4, 5, 6, 7$ each one row higher to $i - 1$ yielding A_3. Hence, the evaluation process is able to add the local partial derivatives $c_{10} = \frac{\partial v_4^2}{\partial v_3^2} = 1$ and $c_{11} = \frac{\partial v_4^2}{\partial v_2^2} = -1$ as the contribution of the statement $v_4^2 = v_3^2 - v_{2,2}$ that results in A_4.

$$
\begin{array}{ccccccc}
v_1^1 & & & & & & \\
0 & v_2^1 & & & & & \\
c_9 & c_3 & v_4^1 & & & & \\
0 & 0 & c_5 & v_5^1 & & & \\
0 & 0 & c_6 & 0 & v_6^1 & & \\
0 & 0 & 0 & c_7 & c_8 & v_3^2 & \\
0 & 0 & 0 & 0 & 0 & 0 & v_\cdot
\end{array}
\qquad A_3
$$

$$
\begin{array}{ccccccc}
v_1^1 & & & & & & \\
0 & v_2^1 & & & & & \\
c_9 & c_3 & v_4^1 & & & & \\
0 & 0 & c_5 & v_5^1 & & & \\
0 & 0 & c_6 & 0 & v_6^1 & & \\
0 & 0 & 0 & c_7 & c_8 & v_3^2 & \\
0 & 0 & 0 & 0 & c_{10} & c_{11} & v_4^2
\end{array}
\qquad A_4
$$

Thus, we eliminate rows v_4^1 and v_3^2 and get A_5, where $c_{12} = c_9 \cdot c_5$, $c_{13} = c_3 \cdot c_5$, $c_{14} = c_9 \cdot c_6$, and $c_{15} = c_3 \cdot c_6$. Updating now A_5 yields A_6, where the last two rows are freed to be used again.

$$
\begin{array}{cccccccc}
v_1^1 & & & & & & & \\
0 & v_2^1 & & & & & & \\
\circledcirc & \circledcirc & v_\cdot & & & & & \\
c_{12} & c_{13} & \circledcirc & v_5^1 & & & & \\
c_{14} & c_{15} & \circledcirc & 0 & v_6^1 & & & \\
0 & 0 & 0 & \circledcirc & \circledcirc & v_\cdot & & \\
0 & 0 & 0 & c_{16} & c_{10} & \circledcirc & v_4^2 &
\end{array}
$$

A_5

$$
\begin{array}{cccccccc}
v_1^1 & & & & & & & \\
0 & v_2^1 & & & & & & \\
c_{12} & c_{13} & v_5^1 & & & & & \\
c_{14} & c_{15} & 0 & v_6^1 & & & & \\
0 & 0 & c_{16} & c_{10} & v_4^2 & & & \\
0 & 0 & 0 & 0 & 0 & v_\cdot & & \\
0 & 0 & 0 & 0 & 0 & 0 & v_\cdot &
\end{array}
$$

A_6

The last evaluation step yields A_7 *by adding the local partial derivatives* $c_{17} = \frac{\partial v_5^2}{\partial v_4^2} = \cos(v_4^2)$ *resp.* $c_{18} = \frac{\partial v_6^2}{\partial v_4^2} = \exp(v_4^2)$ *as the contribution of the statement* $v_5^2 = \sin(v_4^2)$ *resp.* $v_6^2 = \exp(v_4^2)$ *to* A_6*. Finally, the elimination of rows of* v_5^1, v_6^1, *and* v_4^2 *results in* A_8 *containing the Jacobian of our example function for* $n = 2$ *as follows.*

$$
\begin{aligned}
f'_{1,1} &= c_{19} = c_{17} \cdot (c_{16} \cdot c_{12} + c_{10} \cdot c_{14}) & \qquad f'_{1,2} &= c_{20} = c_{17} \cdot (c_{16} \cdot c_{13} + c_{10} \cdot c_{15}) \\
f'_{2,1} &= c_{21} = c_{18} \cdot (c_{16} \cdot c_{12} + c_{10} \cdot c_{14}) & \qquad f'_{2,2} &= c_{22} = c_{18} \cdot (c_{16} \cdot c_{13} + c_{10} \cdot c_{15})
\end{aligned}
$$

$$
\begin{array}{cccccccc}
v_1^1 & & & & & & & \\
0 & v_2^1 & & & & & & \\
c_{12} & c_{13} & v_5^1 & & & & & \\
c_{14} & c_{15} & 0 & v_6^1 & & & & \\
0 & 0 & c_{16} & c_{10} & v_4^2 & & & \\
0 & 0 & 0 & 0 & c_{17} & v_5^2 & & \\
0 & 0 & 0 & 0 & c_{18} & 0 & v_6^2 &
\end{array}
$$

A_7

$$
\begin{array}{cccccccc}
v_1^1 & & & & & & & \\
0 & v_2^1 & & & & & & \\
\circledcirc & \circledcirc & v_\cdot & & & & & \\
\circledcirc & \circledcirc & 0 & v_\cdot & & & & \\
\circledcirc & \circledcirc & \circledcirc & \circledcirc & v_\cdot & & & \\
c_{19} & c_{20} & 0 & 0 & \circledcirc & v_5^2 & & \\
c_{21} & c_{22} & 0 & 0 & \circledcirc & 0 & v_6^2 &
\end{array}
$$

A_8

2.6.2 Iterative Sparsity Exploitation of Extended Jacobians

Sparsity of extended Jacobians can also be exploited iteratively. Therefore, the symbolic step on bit pattern in iterative mode has to keep the given memory limitation interpreted as the number of rows q defined by Equation (2.23), which represents the upper bound for row number of bit pattern proven by Lemma 2.7. The proof idea bases on the assumption that $Mem(BP) \leq Mem(C')$ with $\mu_I \leq \mu_F$.

Hence, given a memory bound of $M = 21 \cdot \mu_F$ bits a bit pattern of at most seven rows as

$$q \leq \frac{1 + \sqrt{1 + 8 \cdot 21}}{2} = \frac{14}{2} = 7 \tag{2.22}$$

fits into the memory, which is also the case for the extended Jacobian discussed previously in Example 2.7.

Lemma 2.7. *Let M represent the available memory in bits. The bit pattern BP can have at most*

$$q = \frac{1 + \sqrt{1 + \frac{8 \cdot M}{\mu_F}}}{2} \tag{2.23}$$

rows.

Proof. From Equation (2.10) it follows that

$$Mem(BP) = \sum_{j=1}^{q} (\left\lceil \frac{j}{\mu_I} \right\rceil + 1) \cdot \mu_I \leq \sum_{j=1}^{q} (j-1) \cdot \mu_F = \frac{q \cdot (q-1)}{2} \cdot \mu_F \quad .$$

Hence, we set

$$\frac{q \cdot (q-1)}{2} \cdot \mu_F = M \Leftrightarrow q^2 - q - \frac{2 \cdot M}{\mu_F} = 0 \quad .$$

The binomial formula yields the following two solutions of the resulting polynomial above

$$q = \frac{1 \pm \sqrt{1 + \frac{8 \cdot M}{\mu_F}}}{2} \quad ,$$

where $1 + \frac{8 \cdot M}{\mu_F}$ is a positive number ≥ 1. □

Example 2.8 illustrates the iterative symbolic row elimination on bit pattern corresponding to those performed on extended Jacobians of Example 2.7. The resulting memory pattern is used in Example 2.9 to accumulate the Jacobian on the resulting CRS. Let us consider the fourth row of B_1 related to v_4^1 with $B_1(4,1) = 6 = 2^1 + 2^2$, where $B_1(4,2)$ stores the number of nonzeros of row 4 as $B_1(4,2) = 2$. The elimination of row v_3^1 yields B_2 containing one fill-in as $B_2(4,1) = 3 = 2^0 + 2^1$ and $B_2(4,2) = 3$ that increases the amount of spots of row 4 to three. Now, we update B_2 and get B_3. Let us consider again row 4 that is moved into row 3. Its two nonzeros are moved into row 3 yielding $B_3(3,1) = 3$ and $B_3(3,2) = 2$. Hence, row 5 can also be moved into row 4. Here we have to be careful with overwriting the entry $B_3(4,2) = 3$ with $B_3(5,3) = 1$, since otherwise we lose the correct (maximum) number of spots of row 4 in this iteration. Thus, we save this value as the current largest spot size of row 4 in $L[4] = 3$ for $L = L(B2)$ before we overwrite it, where L is a integer vector of length q.

Example 2.8. *We illustrate here the iterative symbolic elimination on bit pattern of the extended Jacobian of Example 2.7. Analogous to the extended Jacobian version, we assume a memory limit of seven rows as computed in Equation (*2.22*) and four bit integer as $\mu_I = 4$. Thus, the evaluation process yields the bit pattern B_1 corresponding to the extended Jacobian A_1 of Example 2.7. Elimination of row v_3^1 yields B_2. The update process copies rows $i = 4, 5, 6, 7$ to $i - 1$ yielding B_3 as follows.*

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
3 & 2 & v_3^1 & \\
6 & 2 & v_4^1 & \\
8 & 0 & 1 & v_5^1 \\
8 & 0 & 1 & v_6^1 \\
0 & 3 & 2 & v_3^2
\end{array}
\qquad B_1
$$

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
0 & 2 & v_3^1 & \\
3 & 3 & v_4^1 & \\
8 & 0 & 1 & v_5^1 \\
8 & 0 & 1 & v_6^1 \\
0 & 3 & 2 & v_3^2
\end{array}
\qquad B_2
$$

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
3 & 2 & v_4^1 & \\
4 & 1 & v_5^1 & \\
4 & 0 & 1 & v_6^1 \\
8 & 1 & 2 & v_3^2 \\
0 & 0 & 0 & v_\cdot
\end{array}
\qquad B_3
$$

Thereby, we have

$$L(B_1) = [0,0,2,2,1,1,2], \quad L(B_2) = [0,0,2,3,1,1,2], \quad \textit{and} \quad L(B_3) = [0,0,2,3,1,2,2] \quad .$$

Now, the evaluation process yields B_4 by adding the contribution of the statement $v_4^2 = v_3^2 - v_{2,2}$. Thus, we eliminate rows v_4^1 and v_3^2 and get B_5. Updating B_5 yields B_6, where the last two rows are freed to be used again.

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
3 & 2 & v_4^1 & \\
4 & 1 & v_5^1 & \\
4 & 0 & 1 & v_6^1 \\
8 & 1 & 2 & v_3^2 \\
0 & 3 & 2 & v_4^2
\end{array}
\qquad B_4
$$

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
0 & 2 & v_3^1 & \\
3 & 3 & v_5^1 & \\
3 & 0 & 3 & v_6^1 \\
0 & 0 & 2 & v_\cdot \\
8 & 1 & 3 & v_4^2
\end{array}
\qquad B_5
$$

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
3 & 2 & v_5^1 & \\
3 & 2 & v_6^1 & \\
12 & 0 & 2 & v_4^2 \\
0 & 0 & 0 & v_\cdot \\
0 & 0 & 0 & v_\cdot
\end{array}
\qquad B_6
$$

Thereby, we have

$$L(B_4) = [0,0,2,3,1,2,2], \quad L(B_5) = [0,0,2,3,3,2,3], \quad \textit{and} \quad L(B_6) = [0,0,2,3,3,2,3] \quad .$$

The last evaluation step yielding B_7 followed by elimination of rows v_5^1, v_6^1, and v_4^2 results in B_8 denoting the eliminated bit pattern of our example function for $n = 2$.

$$
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
3 & 2 & v_5^1 & \\
3 & 2 & v_6^1 & \\
12 & 0 & 2 & v_4^2 \\
0 & 1 & 1 & v_5^2 \\
0 & 1 & 1 & v_6^2
\end{array}
\qquad
\begin{array}{cccc}
0 & 0 & v_1^1 & \\
0 & 0 & v_2^1 & \\
0 & 2 & v_{\cdot,\cdot} & \\
0 & 2 & v_{\cdot,\cdot} & \\
0 & 0 & 4 & v_{\cdot,\cdot} \\
3 & 0 & 3 & v_5^2 \\
3 & 0 & 3 & v_6^2
\end{array}
$$

B_7 B_8

Thereby, we have

$$L(B_7) = [0,0,2,3,3,2,3], \quad \textit{and} \quad L(B_8) = [0,0,2,3,4,3,3] \quad .$$

In general, all three evaluation, elimination, and update processes have to take care of the right number of row spots over the entire iterations in symbolic mode to guarantee the correct memory pattern needed in the accumulation mode on CRS. Figure 2.31 illustrates the entire iterative symbolic elimination process via symbolic elimination of rows of bit pattern in σ-order. Thereby, k denotes the iteration index such that v_j^k indicates the execution of the statement v_j in kth evaluation step. Furthermore, the set of eliminatable rows in iteration k is denoted by Z_k.

As shown before by an example, the detection of memory usage of CRS of our example function for $n = 2$ needs three iterations in total. The last iteration yields B_8 with the total number of fourteen spots as the sum over spots of rows stored in L. Knowing this the corresponding CRS can be statically allocated, which is supposed to be initialized in the second evaluation of F in the accumulation mode with real values, which is shown in Figure 2.32. It is worth mentioning here that in iterative mode, we do not care about the ordering of the nonzero elements in CRS. Consequently, the initialization as well as elimination processes are free to put the values in arbitrary empty spots in range of the corresponding rows. However, one side effect of this is the linear index search over row entries. Its impact on runtime of sparse Jacobian accumulation on CRS representation of extended Jacobians have already been discussed with the test cases in Section 2.4.2. We note that for consistency reasons we decided to use the linear search algorithm overall in DALG. However, the implementation of more efficient algorithms is the focus of ongoing implementation activity on DALG.

In the following we show all those algorithms described in Section 2.2 and Section 2.4 that have to be modified to make them work in iterative mode. Here, we assume that q represents the maximum number of statically allocatable rows for bit pattern according to Equation (2.23), where again L is supposed to be an initially zero integer vector of length q. Thereby,

$$L(i) = \max_{k=1}^{\nu}(\tilde{BP}_k\,(i, \left\lceil \frac{i}{\mu_I} \right\rceil + 1) \quad \text{with} \quad \tilde{BP}_k = BP_k - Z_k$$

represents the maximum number of spots of row $i = 1, \cdots, q$ over all ν iterations. $\tilde{BP}_k$ denotes the eliminated bit pattern resulted from the elimination of intermediate rows Z_k at the iteration $k \in \{1, \ldots, \nu\}$ on BP_k, which is supposed to be initialized in the kth evaluation step. In the following algorithms we assume $p = |Z_k|$ to denote the number of intermediate rows of the kth iteration.

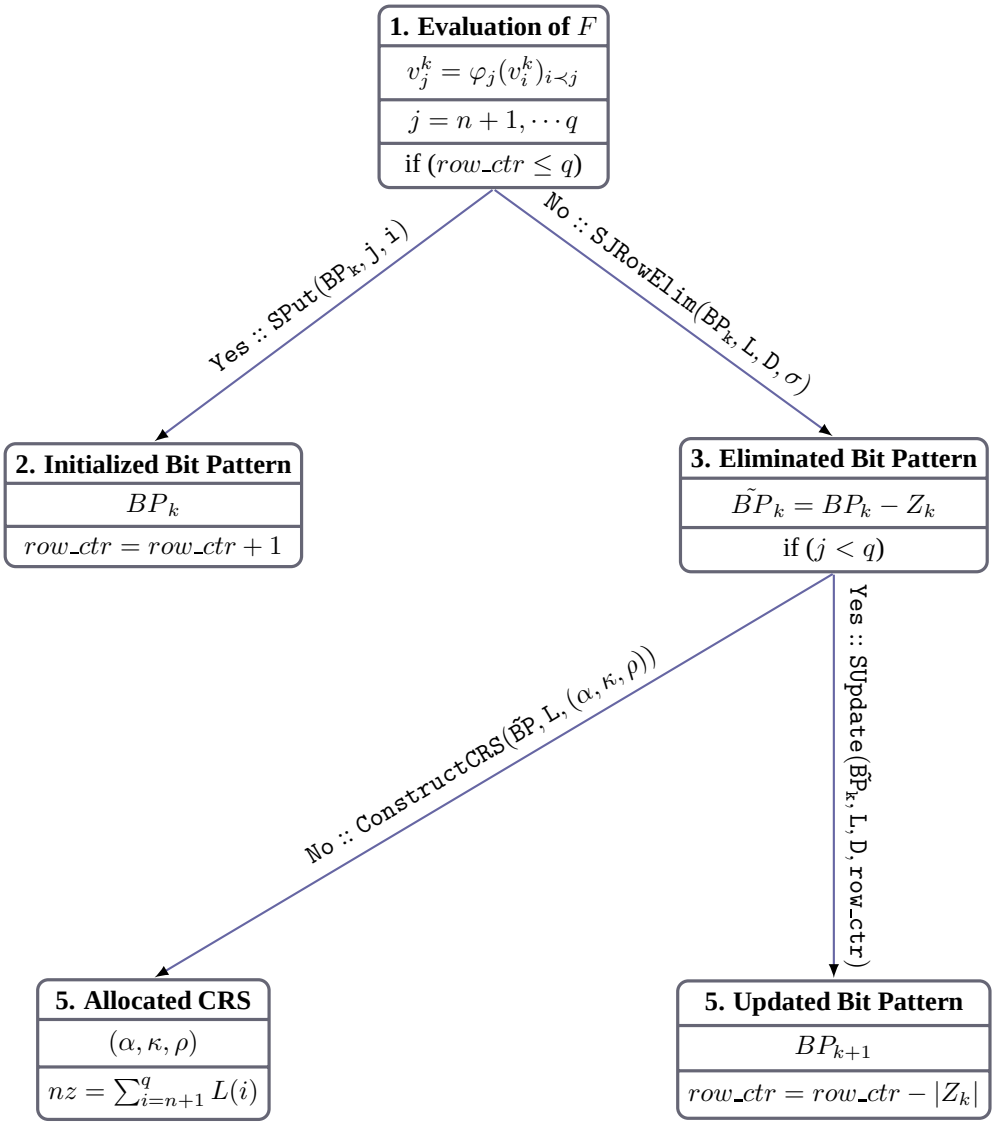


Figure 2.31: Iterative Symbolic Elimination Process on Bit Pattern.

Algorithm 2.17 (`JSRowElim` $(\texttt{BP}, \texttt{L}, \texttt{D}, \sigma)$: Symbolic Row Elimination)**.**

Require: Bit pattern BP, integer vector L, and Boolean vector D of length q.
Ensure: BP after elimination of all intermediate columns in σ ordering.

1: **for** $j = \sigma(1)$ **to** $\sigma(p)$ **do**
2: `SRowElim` $(\texttt{BP}, \texttt{D}, \texttt{j})$
3: $D(j) = true$
4: **for** $i = 1$ **to** q **do**
5: $L(i) = \max(\, L(i), BP(i, \left\lceil \frac{i}{\mu_I} \right\rceil + 1)\,)$
6: **end for**
7: **end for**

Algorithm 2.18 (`SUpdate`$(\texttt{BP}, \texttt{L}, \texttt{D}, \texttt{row\_ctr})$: Update Bit Pattern)**.**

Require: Bit pattern BP, integer vector L, Boolean vector D of length q, and row counter row_ctr.
Ensure: Updated bit pattern BP and L.

1: **for** $j = n + 1$ **to** q **do**
2: **if** $D(j) == true$ **then**
3: $nz = 0$
4: **for** $i = j + 1$ **to** q **do**
5: **if** $(D(i) == false)$ **then**
6: **for** $k = 1$ **to** $\left\lceil \frac{i}{\mu_I} \right\rceil$ **do**
7: **for** $m = 0$ **to** $\mu_I - 1$ **do**
8: $l = (k-1) \cdot \mu_I + m$
9: **if** $BP(i,k) \& 2^m == 1$ **and** $D(l) == true$ **then**
10: $nz = nz + 1$
11: **end if**
12: **end for**
13: $BP(j,k) = BP(i,k)$
14: $BP(i,k) = 0$
15: **end for**
16: $BP(j, \left\lceil \frac{j}{\mu_I} \right\rceil + 1) = nz$
17: $BP(i, \left\lceil \frac{i}{\mu_I} \right\rceil + 1) = 0$
18: # Replacing $i \prec k$ with $j \prec k$
19: **for** $k = i + 1$ **to** q **do**
20: **if** $D(k) == false$ **and** $BP(k, \left\lceil \frac{i}{\mu_I} \right\rceil) \& 2^{(i-1)\%\mu_I} == 1$ **then**
21: $BP(k, \left\lceil \frac{j}{\mu_I} \right\rceil) = BP(k, \left\lceil \frac{j}{\mu_I} \right\rceil) \mid 2^{(j-1)\%\mu_I}$
22: $BP(k, \left\lceil \frac{i}{\mu_I} \right\rceil) = BP(k, \left\lceil \frac{i}{\mu_I} \right\rceil) - 2^{(i-1)\%\mu_I}$
23: **end if**
24: **end for**
25: **end if**
26: **end for**
27: $L(j) = \max(L(j), nz)$
28: $row\_ctr = j$
29: $D(i) = true$

30: **end if**
31: **end for**

Algorithm 2.19 (`ConstructCRS`$(\texttt{BP}, \texttt{L}, (\alpha, \kappa, \rho))$: CRS Construction)**.**

Require: Bit pattern BP, integer vector L of length q.
Ensure: Initialized CRS (α, κ, ρ).

1: `free`$(\texttt{BP})$
2: $c = 1$
3: `allocate`$(\rho, \texttt{q} + 1)$
4: **for** $i = 1$ **to** n **do**
5: $\rho(i) = 1$
6: **end for**
7: **for** $i = n + 1$ **to** q **do**
8: $\rho(i) = c$
9: $c = c + L(i)$
10: **end for**
11: `allocate`$(\alpha, \texttt{c})$
12: `allocate`$(\kappa, \texttt{c})$
13: **for** $i = 1$ **to** $c - 1$ **do**
14: $\alpha(i) = 0$
15: $\kappa(i) = 0$
16: **end for**
17: $\rho(q + 1) = c - 1$

After termination of ν symbolic row eliminations Algorithm 2.19 can be used to construct the resulting CRS for given L and q, where α, κ and ρ are of length $\sum_{i=n+1}^{q} L(i)$ and $q + 1$, respectively. Thereby, the routine call `free`$(\texttt{BP})$ in line 1 indicates that the memory allocated for BP is freed and hence the entire available memory M can be used to store CRS. However, the construction step assumes that the resulting CRS fits into M, that is, $Mem(CRS) \leq M$ according to Equation (2.8), which we consider reasonable. α and ρ are initialized to zero as shown in lines 13–16. Element i with $i = 1, \cdots, n$ [$i = n + 1, \cdots, q$] of ρ is initialized to one [the position of the first nonzero element of row i in α vector as shown in line 5 [8].

Once CRS is constructed Algorithm 2.14 can again be used to insert partial derivatives into CRS. Here, it has to be assured that exactly the same iteration points are taken as in symbolic mode. This is essential because inconsistency in iterations may cause different memory requirements. Consequently, the evaluation process in accumulation mode has to jump into the elimination step at the same line in the SAC of F as done in symbolic mode. However, this can be simply done by taking $row\_ctr < q$ as condition in evaluation process of both symbolic and accumulation modes as shown in Figures 2.31 resp. 2.32. Thus, at every elimination step k the resulting CRS denoted by $(\alpha, \kappa, \rho)_k$ is transformed into the eliminated version $(\tilde{\alpha}, \tilde{\kappa}, \tilde{\rho})_k$ via row elimination. After the last iteration the Jacobian can be extracted using Algorithm 2.9. Otherwise, Algorithm 2.20 is provided to update CRS before proceeding with the next iteration of the evaluation process. Example 2.9 illustrates the Jacobian accumulation by row elimination of our example function for $n = 2$ step by step on the corresponding CRS of the extended Jacobians of Example 2.7. The respective symbolic eliminations that yield the memory usage of CRS have been illustrated in Example 2.8.

Algorithm 2.20 (`Update`$((\alpha, \kappa, \rho), \texttt{D})$: Update CRS)**.**

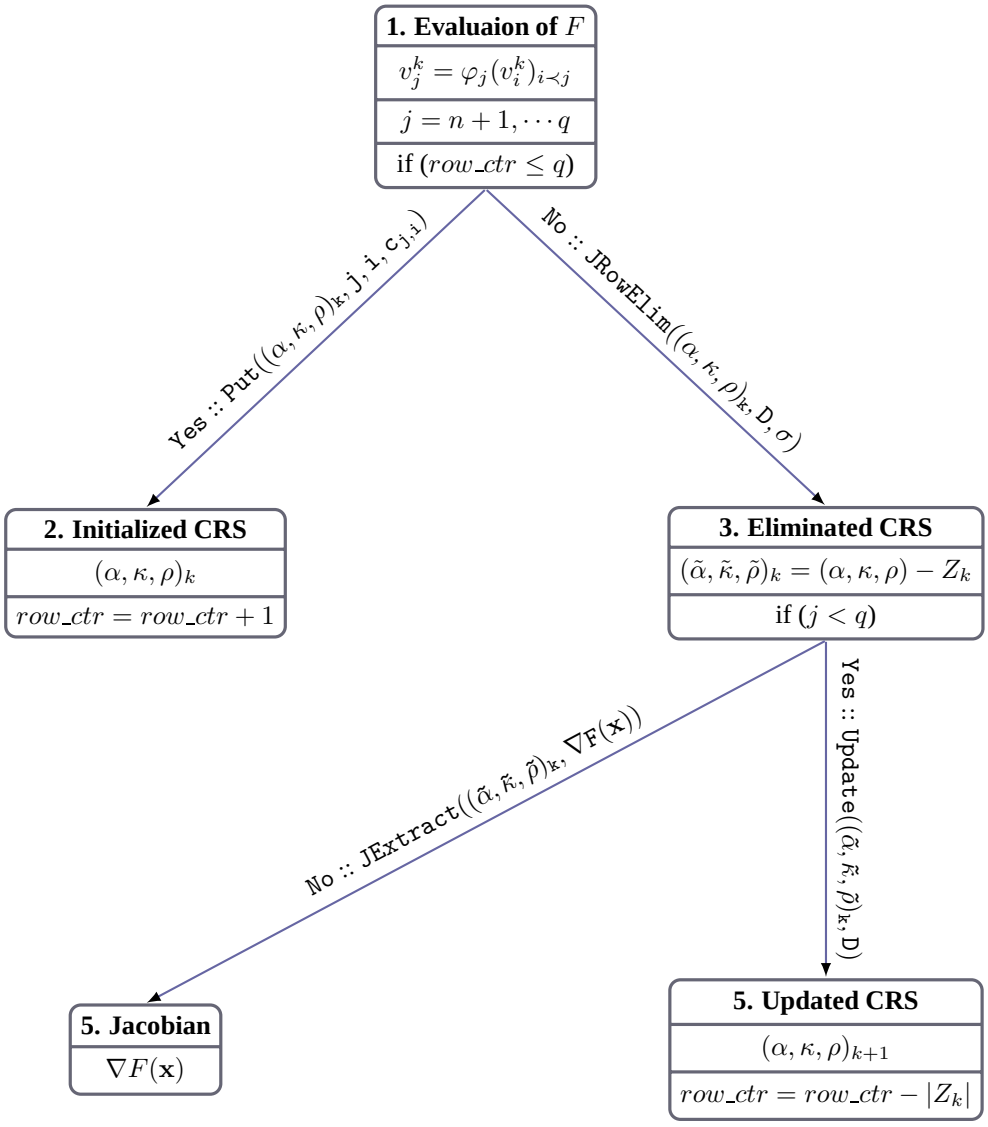


Figure 2.32: Iterative Elimination Process on CRS.

Require: (α, κ, ρ), Boolean vector D of length q.
Ensure: Updated (α, κ, ρ).

```
1:  for j = n + 1 to q do
2:   if D(j) == true then
3:    c = ρ(j)
4:    for i = j + 1 to q do
5:     if (D(i) == false) then
6:      for l = ρ(i) to ρ(i + 1) − 1 do
7:       α(c) = α(l)
8:       κ(c) = κ(l)
9:       α(l) = κ(l) = 0
10:      c = c + 1
11:     end for
12:     # Replacing i ≺ k with j ≺ k
13:     for k = i + 1 to q do
14:      if (D(k) == false) then
15:       l = Find((α, κ, ρ), k, i)
16:       if l > 0 and κ(l) == i then
17:        κ(l) = j
18:       end if
19:      end if
20:     end for
21:    end if
22:   end for
23:  end if
24: end for
```

Example 2.9. *The following illustrates the iterative accumulation of the Jacobian of our example function for $n = 2$ on the CRS, which is generated based on the bit pattern B_8 of Example 2.8. Furthermore, we assume that there is enough memory for the resulting CRS of totally fifteen nonzeros.*

$$\begin{aligned}\alpha =&(\overbrace{0,0}^{\texttt{row 3}},\overbrace{0,0,0}^{\texttt{row 4}},\overbrace{0,0,0,0}^{\texttt{row5}},\overbrace{0,0,0}^{\texttt{row 6}},\overbrace{0,0,0}^{\texttt{row 7}})\\ \kappa =&(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)\\ \rho =&(1,1,1,3,6,10,13,16)\end{aligned}$$

First evaluation of F yields the following CRS representation of the extended Jacobian A_1 shown in Example 2.7, which we use here for better illustration.

$$\begin{aligned}\alpha_1 =&(\overbrace{c_1,c_2}^{\texttt{row 3}},\overbrace{c_3,c_4,0}^{\texttt{row 4}},\overbrace{c_5,0,0,0}^{\texttt{row 5}},\overbrace{c_6,0,0}^{\texttt{row 6}},\overbrace{c_7,c_8,0}^{\texttt{row 7}})\\ \kappa_1 =&(1,2,2,3,0,4,0,0,0,4,0,0,5,6,0)\\ \rho_1 =&(1,1,1,3,6,10,13,16)\end{aligned}$$

Elimination of row 3 related to v_3^1 yields $(\tilde{\alpha}_1, \tilde{\kappa}_1, \tilde{\rho}_1)$ that corresponds to A_2. Thereby, c_3 and c_9 denote an absorption and fill-in, respectivel. Fill-in, fill-out as well as absorptions are denoted by bold letters.

Thereby, the dependence of row 4 on row 3 is given by $c_4 = \frac{\partial v_4^1}{\partial v_3^1} = 1$.

$$\begin{aligned}
\tilde{\alpha}_1 =&(\overbrace{\mathbf{0},\mathbf{0}}^{\texttt{row 3}},\overbrace{\mathbf{c_3},\mathbf{0},\mathbf{c_9}}^{\texttt{row 4}},\overbrace{c_5,0,0,0}^{\texttt{row 5}},\overbrace{c_6,0,0}^{\texttt{row 6}},\overbrace{c_7,c_8,0}^{\texttt{row 7}})\\
\tilde{\kappa}_1 =&(\mathbf{0},\mathbf{0},\mathbf{2},\mathbf{0},\mathbf{1},4,0,0,0,4,0,0,5,6,0)\\
\tilde{\rho}_1 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

The update process copies rows $i = 4,5,6,7$ *into row* $i-1$. *Thus, we get* $(\alpha_2,\kappa_2,\rho_2)$ *corresponding to* A_3 *as follows.*

$$\begin{aligned}
\alpha_2 =&(\overbrace{c_3,c_9}^{\texttt{row 3}},\overbrace{c_5,0,0}^{\texttt{row 4}},\overbrace{c_6,0,0,0}^{\texttt{row 5}},\overbrace{c_7,c_8,0}^{\texttt{row 6}},\overbrace{0,0,0}^{\texttt{row 7}})\\
\kappa_2 =&(2,1,3,0,0,3,0,0,0,4,5,0,0,0,0)\\
\rho_2 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

Hence, the evaluation process adds the contribution of the statement $v_4^2 = v_3^2 - v_{2,2}$ *to* $(\alpha_2,\kappa_2,\rho_2)$ *yielding the following CRS corresponding to* A_4.

$$\begin{aligned}
\alpha_2 =&(\overbrace{c_3,c_9}^{\texttt{row 3}},\overbrace{c_5,0,0}^{\texttt{row 4}},\overbrace{c_6,0,0,0}^{\texttt{row 5}},\overbrace{c_7,c_8,0}^{\texttt{row 6}},\overbrace{c_{10},c_{11},0}^{\texttt{row 7}})\\
\kappa_2 =&(2,1,3,0,0,3,0,0,0,4,5,0,5,6,0)\\
\rho_2 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

The elimination of rows 3 and 6 related to v_4^1 *and* v_3^2 *yields the corresponding CRS of* A_5 *as follows. Thereby, the dependency of rows 5,6 on 3 and row 7 on 6 is represented by the partials* c_5, c_6 *and* c_{11}, *respectively.*

$$\begin{aligned}
\tilde{\alpha}_2 =&(\overbrace{\mathbf{0},\mathbf{0}}^{\texttt{row 3}},\overbrace{\mathbf{0},\mathbf{c_{12}},\mathbf{c_{13}}}^{\texttt{row 4}},\overbrace{\mathbf{0},\mathbf{c_{14}},\mathbf{c_{15}},\mathbf{0}}^{\texttt{row 5}},\overbrace{\mathbf{0},\mathbf{0},0}^{\texttt{row 6}},\overbrace{\mathbf{c_{10}},\mathbf{0},\mathbf{c_{16}}}^{\texttt{row 7}})\\
\tilde{\kappa}_2 =&(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{0},\mathbf{1},\mathbf{2},0,\mathbf{0},\mathbf{0},0,\mathbf{5},\mathbf{0},\mathbf{4})\\
\tilde{\rho}_2 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

Updating $(\tilde{\alpha}_2,\tilde{\kappa}_2,\tilde{\rho}_2)$ *yields the following CRS corresponding to* A_6.

$$\begin{aligned}
\alpha_3 =&(\overbrace{c_{12},c_{13}}^{\texttt{row 3}},\overbrace{c_{14},c_{15},0}^{\texttt{row 4}},\overbrace{c_{10},c_{16},0,0}^{\texttt{row 5}},\overbrace{0,0,0}^{\texttt{row 6}},\overbrace{0,0,0}^{\texttt{row 7}})\\
\kappa_3 =&(1,2,1,2,0,4,3,0,0,0,0,0,0,0,0)\\
\rho_3 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

The last evaluation step adds entries c_{17} *and* c_{18} *to row 6 and 7, respectively yielding the following CRS of* A_7.

$$\begin{aligned}
\alpha_3 =&(\overbrace{c_{12},c_{13}}^{\texttt{row 3}},\overbrace{c_{14},c_{15},0}^{\texttt{row 4}},\overbrace{c_{10},c_{16},0,0}^{\texttt{row 5}},\overbrace{c_{17},0,0}^{\texttt{row 6}},\overbrace{c_{18},0,0}^{\texttt{row 7}})\\
\kappa_3 =&(1,2,1,2,0,4,3,0,0,5,0,0,5,0,0)\\
\rho_3 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

Finally, eliminating rows 3,4,5 related to v_5^1, v_6^1, and v_4^2, respectively results in $(\tilde{\alpha}_3, \tilde{\kappa}_3, \tilde{\rho}_3)$ corresponding to A_8 with row-wise Jacobian entries c_{19}, c_{20} and c_{21}, c_{22}.

$$\begin{aligned}
\tilde{\alpha}_3 =&(\overbrace{\mathbf{0},\mathbf{0}}^{\texttt{row 3}},\overbrace{\mathbf{0},\mathbf{0},0}^{\texttt{row 4}},\overbrace{\mathbf{0},\mathbf{0},0,0}^{\texttt{row 5}},\overbrace{\mathbf{0},\mathbf{c_{19}},\mathbf{c_{20}}}^{\texttt{row 6}},\overbrace{\mathbf{0},\mathbf{c_{21}},\mathbf{c_{22}}}^{\texttt{row 7}})\\
\tilde{\kappa}_3 =&(\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{0},0,0,\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{0},\mathbf{1},\mathbf{2})\\
\tilde{\rho}_3 =&(1,1,1,3,6,10,13,16)
\end{aligned}$$

2.6.3 Numerical Results

In the following we present first numerical results on Jacobian accumulation by row elimination on extended Jacobians as well as the respective CRS representations in iterative fashion using DALG on both test cases Bratu and Heat described in Section 2.4.2. Henceforth, we use the terminologies IRM and IFM to denote iterative Jacobian accumulation on DEJ or CRS with local application of row elimination in forward and reverse ordering, respectively. In this context, we use the available amount of memory M and the resulting conservative number of allocatable rows q by Equation (2.21) interchangeably. Furthermore, we will compare the runtime behavior of iterative approach by focusing on the impact of the size of the *active block* using both DEJ and CRS. An active block of an extended Jacobian consists of all rows for which memory is allocated; elimination process is supposed to act on this part. In other words, we take a part of totally allocatable memory M and try to accumulate iteratively the target Jacobian on it.

We achieve this, for instance, by choosing

$$M_\delta = \delta \cdot M \quad \xrightarrow{Equation\ (2.21)} \quad q_\delta = \frac{1+\sqrt{1+\frac{8\cdot M_\delta}{\mu_F}}}{2} \quad ,$$

where M_δ [q_δ] denotes the memory usage [static row size] of the active block for $0 \leq \delta \leq 1$. Obviously, the entire available memory is allocatable for $\delta = 1$, which corresponds to the non-iterative fashion with the difference that memory is freed after elimination step by the update process enabling further evaluation steps. In case of $\delta = 0$ DALG performs assignment level elimination (ALE), meaning that every execution of an assignment in the program of F is succeeded by an elimination and update step.

Bratu Problem

Our focus is again on the accumulation of the Jacobian of the Bratu function of Listing 2.1. Figure 2.33 (a) and (b) compare runtimes of DALG in iterative mode on DEJ and CRS, respectively, with their non-iterative serial counterparts. As one can see the reverse mode performs better than the forward mode in both non-iterative (GRM) and iterative (IRM) modes on both DEJ and CRS.

To clarify this, let us consider Table 2.4, which presents the runtime of DALG in the three modes non-iterative serial, non-iterative parallel (with eight threads), and iterative modes. We note again that the capability of the first two modes is restricted by the memory bandwidth.

As also discussed in Section 2.4.2 the non-iterative reverse mode (GRM) is about a factor of $2.9 \approx \frac{3162}{1082}$ faster than its forward counterpart (GFM). The runtime difference was conjectured to be caused mostly by the difference in the respective number of multiplications as $1.89 = \frac{362600}{191688}$ along with some cache effects.

Considering now iterative mode using assignment level elimination, we observe analog runtime difference between forward and reverse elimination on DEJ as $1.88 = \frac{47}{25}$, which is even closer to the factor 1.89 achieved by the multiplication difference above[5]. However, this is not really surprising, since the resulting elimination sequence in forward/reverse at the end of IJA is not different than the non-iterative

[5]We believe the reason for this to be the better cache behavior during ALE.

one as there is no dependence between the results of assignments (r[i][j]) in Bratu function as shown in line 13 of Listing 2.1. One can easily figure out that every execution of an assignment contributes directly to a row of the target Jacobian. Hence, it is not surprising that reverse elimination is faster than forward mode. Thus, the iterative approach seems to accelerate DJARE and SJARE in both forward and reverse orderings considerably. In particular, IRM in context of DJARE [SJARE] is about a factor of roughly $43 \approx \frac{1082}{25}$ [$311 \approx \frac{1556}{5}$] faster than its GRM counter part.

Comparing now the runtimes of IRM by assignment level elimination on DEJ with those of CRS we observe that the latter is about a factor $5 = \frac{25}{5}$ faster that the former. Thus, we observe that assignment level elimination on CRS performs much better than on DEJ in case of Bratu. However, our next test case will show that this behavior may change.

$n = 100$	GFM	GRM	IFM	IRM	LFM (#8)	LRM (# 8)
DEJ	3162	1082	47	25	121	118
CRS	7159	1556	15	5	17	9

Table 2.4: Runtime Measurement Data for Bratu for $n = 100$ in seconds.

The respective memory consumptions of DEJ and CRS in iterative mode are shown in (d). Thereby, the memory consumption of DEJ grows polynomially as opposite to CRS that behaves roughly linear with n. The reason for hight memory consumption of DEJ is because Bratu generates a lot of program variables that can not be eliminated over entire IJA. More precisely, for $n = 100$ we measure that DEJ allocates 2881 MB heap memory. To clarify this let us considering the lines 8–16 and 18–20 of Bratu function. One can easily see that all rows related to r and x both of size $(n-2)^2$ are alive and hence can not be eliminated over iterations inside of bratu. Therefore, all temporary rows generated by the right hand side of the expression of line 12 can be eliminated to be reused later. Hence, updating DEJ yields a matrix with roughly $q \approx 2 * (n-2)^2$ rows. Thus, we get for $n = 100$ and $\mu_F = 8$ Bytes

$$q = 19208 \xrightarrow{Equation\ (2.7)} Mem(DEJ) = \frac{19208 \cdot (19208-1)}{2} \cdot \mu_F \approx 1407\,\text{MByte} \quad .$$

The reason for the difference between allocated memory by DALG and the one calculated above is that DALG has to maintain some meta data in addition to DEJ to perform elimination.

Furthermore, (c) compares runtimes of IRM on DEJ and CRS with their non-iterative parallel counterparts using eight threads denoted by LRM (#8). As one can see runtime of LRM(#8) overtakes that of IRM both on CRS for sufficiently large dimensions. More precisely, for $n = 200$ the latter needs 310 seconds in total as opposed to 200 seconds in the former to accumulate the Jacobian of the Bratu function. We note that the runtime gap may widen for even larger problem sizes.

Based on our results, so far we observed that assignment level elimination performs better in runtime and memory usage as shown in (a) and (b), respectively. Therefore, increasing active block sizes by choosing larger δ seems to scale down the performance. In fact, the same holds for CRS even when the memory increase is not as strong as in case of DEJ. As discussed in Section 2.5.4 main contribution to the speedup of PJA was conjectured to be a side effect of smaller search spaces for dependencies. Hence, high gain in runtime was observed on Bratu even with a single thread compared with the serial version in both forward and reverse ordering as shown in Table 2.3. On the other hand, as shown in (e) IRM on both DEJ and CRS underperforms asymptotically by increasing the size of the active block δ. We believe that the runtime loss is again because of the increase of the size of search space in larger active blocks, which also results in higher memory usage as shown in (f). In (e) we observe also that DEJ perform better than CRS for sufficient large active blocks, which conforms with the runtime in non-iterative mode shown in Figure 2.12. Thus, we believe that combining IJA approach with the parallel one to be very promising. Here, we intend to accumulate the entire available memory during the function evaluation followed by

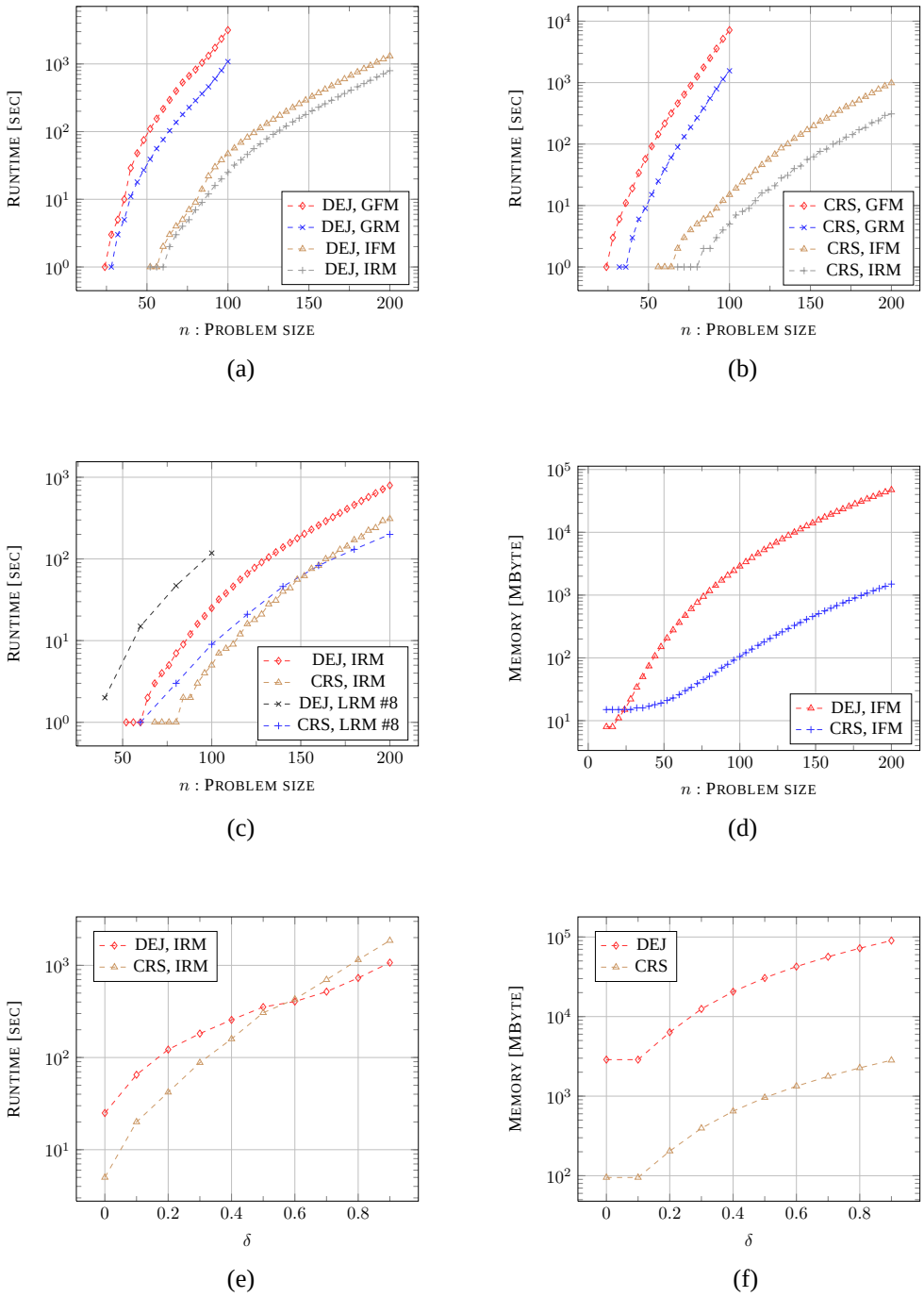


Figure 2.33: Runtime and Memory Results of DALG in Iterative Mode using DEJ and CRS on Bratu.

parallel elimination on sub-matrices. Termination of parallel session is succeeded by the update step to proceed with evaluating the function if necessary. Hence, we hope to achieve better scalability by IJA along with runtime improvement as side effect of smaller search spaces by PJA.

Heat Equation

Figure 2.34 presents numerical results on runtime and memory consumption (b) of IJA by DALG for the computation of the gradient ∇f of f given in line 20 of Listing 2.2. The focus here is only on reverse ordering on DEJ as in Heat DEJ turns out to be almost faster than its CRS counterpart. Therefore, let us consider (c) that compares runtime of DEJ with CRS. Here, we alternate the active block size as discussed on Bratu in Figure 2.33 (e). However, DEJ seems to perform almost better than CRS, despite the fact that it requires more memory by increase in δ. However, the default IJA mode namely assignment level elimination for $\delta = 0.0$ seems to perform and keep the memory consumption low for high dimensions $nx = 100, \ldots, 1000$ with $nt = 100 \cdot nx$.

One can easily figure out that the memory usage of DALG on Heat is much better that on Bratu as discussed previously in Figure 2.33 (d). The reason again lies in the nature of the underlying program of Heat. Here, DEJ consists of a constant factor of nx alive and hence not eliminatable rows over the entire iterative process. Thus, our numerical experiments show that the benefit of IJA depends very much on

	$nx = 100, nt = 10000$		$nx = 200, nt = 20000$		$nx = 300, nt = 30000$	
	Secs.	MByte	Secs.	MByte	Secs.	MByte
IRM ($\delta = 0$)	170	3	3024	4	27796	5
IRM ($\delta < 0$)	11	6	205	17	957	36
ADOLC, GRM	16	273	80	1097	213	2473

Table 2.5: Summary of Measurement Data for Heat using ADOL-C and DALG.

the implementation of the underlying problem as discussed in very detail at the beginning of Section 2.6.

A detailed view on runtime and memory of the iterative mode of DALG is given in Table 2.5. The first row presents runtime (in seconds) and memory (in Megabyte) behavior of IRM by assignment level elimination, that is, $\delta = 0$. Let us now compare its runtime with the global reverse mode AD implementation of ADOL-C[6]. Here, ADOL-C stores the tape on the hard disk. As one can see for input dimension $nx = 100$ and $nt = 10000$ time steps IRM by ALE is roughly $15 \approx \frac{170}{11}$ slower than GRM of ADOL-C. On the contrary, IRM reduces the storage usage by a factor of roughly $45 \approx \frac{273}{6}$. Hence, the gain in memory by IJA is much higher than the loss in runtime. Note that both runtime and storage gaps increase considerably in both cases. We conjecture the reason for the runtime loss of IRM by ALE to lie again in the nature of Heat. Therefore, the reader may easily figure out that roughly $nx \times nt$ assignments (lines 5-6 of Listing 2.2) are performed in the code of f. Hence, roughly $nx \cdot nt$ consecutive elimination and update steps are performed over the entire iterative process, where in the former approximately 5 intermediate rows are eliminated. Hence, we believe the low number of eliminatable intermediates over entire IJA to be the reason for the loss on performance. In order to tackle this problem we aim to perform ALE while adapting the size of active blocks as

$$if(row\_ctr \geq 1.1 \cdot ub) \quad ub = \min(2 \cdot row\_ctr, q) \quad . \tag{2.24}$$

Thus, an assignment in the program of f leads to an elimination and hence update step if the current row counter row_ctr (see Figure 2.31) is at least ten percent greater than the update bound ub, which is assumed to be initially zero. Thereby, the update bound is adjusted by the factor 1.1, which shows so far the best runtime behavior on Heat. To achieve this effect in DALG one has to choose a negative δ.

[6]We use ADOL-C release 2.1.12 available at `http://www.coin-or.org/projects/ADOL-C.xml`

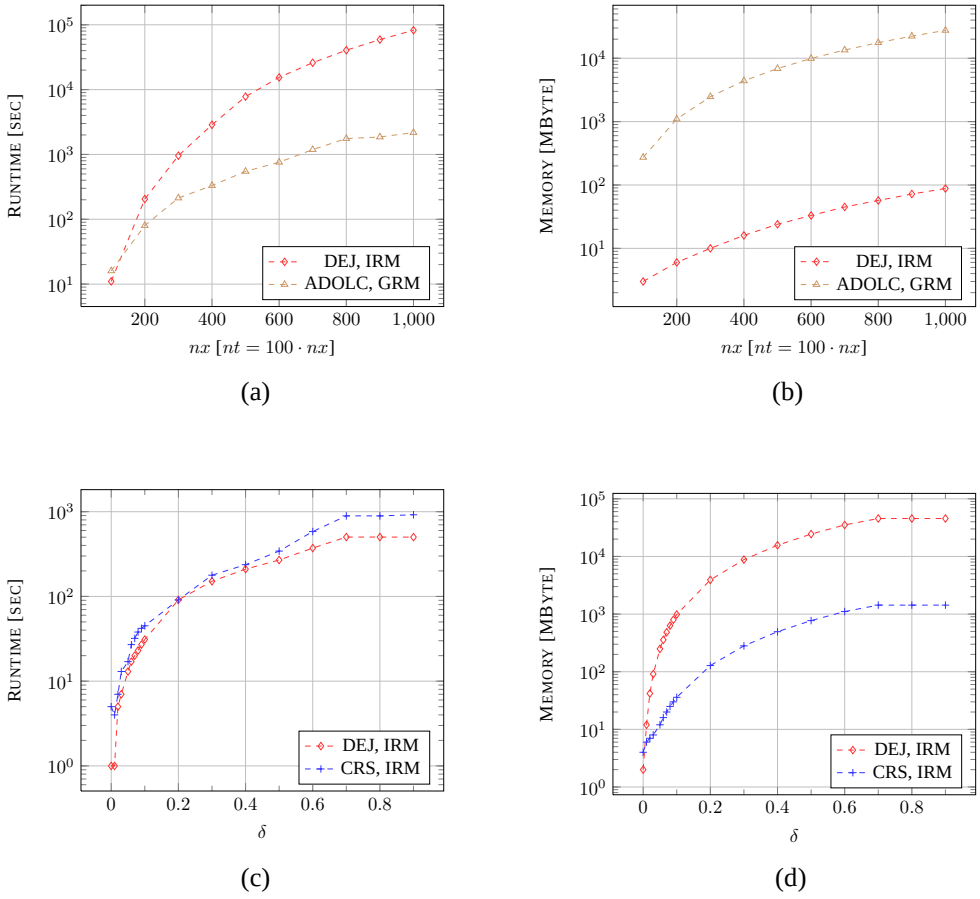


Figure 2.34: Runtime and Memory Results of DALG in Iterative Mode on DEJ and CRS on Heat.

Doing this for the Heat problem as shown in the second line of Table 2.5 we observe high improvement in runtime of IJA with negligible loss of memory. In particular, ∇f for $nx = 100$ and $nt = 10000$ is now computable in 11 seconds instead of 170 seconds by ALE without memory (row counter) adaption. Thus, the runtime gap between IJA by DALG and GRM using ADOL-C is reduced considerably by the former. We note that we use IRM in memory adaptive mode in our plots (a) and (b).

Based on our numerical results, we have considered two different types of problems, namely Bratu and Heat. In the the Bratu case CRS turns out to be much more suitable to face the memory issue, whereas in the case of Heat the opposite is the case. However, applying assignment level elimination seems to be the most memory-friendly mode available in DALG that turns out to be also more efficient in runtime for Bratu but not for Heat. However, in the latter we observed that adapting the size of active blocks at runtime according to Equation (2.24) improves the runtime of IJA on DEJ considerably.

Our numerical results show the potential of IJA as the general purpose approach to tackling the memory problem of the reverse mode AD by minimizing the users intervention. Hence, gradients of even larger dimensions can be accumulated very cheaply in memory using IJA, where the global reverse mode AD would fail. Thus, we consider the observed performance loss compared to the global reverse mode AD provided by ADOL-C as acceptable.

At this point we note that checkpointing strategy would of course solve the memory problem of GRM. Especially in the case of Heat as a linear inverse problem a simple checkpoint using ADOL-C would be enough to accumulate ∇f. However, as motivated at the beginning of this chapter the application of checkpointing requires AD expertise. This may be easy in case of Heat. However, discovering the applicability of checkpointing and adapting it into real world inverse problems [UHP$^+$09] might be a research project on its own.

However, our further investigations will focus in very detail on runtime improvement of IJA. In that context, we plan to implement IJA idea on DAGs for comparisons. Here, the main focus will be on reducing the dynamic effects resulted by the memory allocation and deallocation instructions at runtime. In particular, it is preferable that the elimination of a particular vertex do not necessarily lead to the memory deallocation of that vertex. In this context, a deleted vertex is supposed to kept in memory, so that it can represent in general different (SAC) variables over the entire iterative process. Thus, we hope to benefit from the local dependencies of vertices at each iteration point given as their predecessors. We recapitulate that this was encountered to matter on DEJ and its CRS by increasing the size (q) of the respective matrices. Thus, so far we can not benefit from the entire available memory using DEJ as well as CRS. However, further research is planed to solve this problem and improve the runtime behavior of CRS.

Chapter 3

Detection and Exploitation of Sparsity in Derivative Tensors

3.1 Motivation and Summary of Results

In the following we investigate methods for improving the prediction and exploitation of the sparsity of in general derivative tensors $\nabla^d F$ such as Jacobian ∇F ($d = 1$) and Hessian $\nabla^2 F$ ($d = 2$) of the function F defined by Equation (1.1). Existing compression techniques [CPR74] are based on the knowledge of the nonzeros sparsity pattern of target Jacobians or Hessians. To achieve a better compression, Section 3.2 distinguish between the variable and constant nonzero entries of $\nabla^d F$ as defined by Equation (3.1); the former does not depend on the input values as opposed to the latter, that needs to be computed at runtime. Dynamic algorithms are provided in Section 3.2.2 and Section 3.2.3 for estimating the sparsity pattern and constant entries of $\nabla^d F$, respectively. We note that both are supposed to be sparser than their nonzero counterparts. As two case studies, constants are exploited in the process of Jacobian and Hessian computation in Section 3.2.4 and Section 3.2.5, respectively.

At this point it is worth mentioning that the compression of Jacobian or Hessian matrices is achieved by applying some coloring heuristic to the respective graph representations. The respective coloring problem are known to be NP-complete. The heuristics used in this are provided by the graph coloring package ColPack [GMP05].

Obviously, the generation of the graph of a particular derivative matrix requires only the knowledge about its sparsity pattern. Henceforth, whenever we talk about coloring a matrix or its sparsity pattern we mean coloring the respective graph of that matrix. In the case of constant exploitation we aim to color the variable pattern of both Jacobians and Hessians. Our experimental results in Section 3.2.6 show that constant exploitation performs in terms of achieved colors in context of sparse Jacobian accumulation even on a originally very sparse (nonzero) Jacobian. However, no gain in colors is achieved by constant exploitations in context of sparse Hessian computation considering the objective (scalar) function arising in the context of *Simulated Moving Bed* (SMB) process a model for liquid chromatographic separation described by Gebremedhin et al. [GPW08] as shown in Table 3.2 on page 122.

More precisely, star coloring of the adjacency graph of roughly twelve percent sparser variable Hessian of dimension (34305×34305) yields 14401 colors instead of 12346 when coloring its nonzero pattern. Moreover, the star coloring of the variable pattern underperforms compared with its nonzero counterpart. Note that the computation of constants along with the variable pattern is much more expensive. Walther [Wal08] has shown that the detection of the nonzero sparsity pattern is, in worst case, quadratic in the maximum number of nonzeros per row overall Hessian rows. Thus, the coloring seems

to be the major obstacle toward achieving a better coloring by constant exploitation in context of sparse Hessian computation. At this point we emphasize that the variable Hessian of our test objective function is much sparser than the nonzero one. So far the coloring results do not motivate our high effort in estimating constant Hessian part along with its variable pattern.

Nonetheless, Hessians are ingredients of a lot of numerical applications such as the *inverse medium problem* as introduced in chapter 8 of [NS11] being a large scale PDE-constraint optimization problem. There, the computation of the Hessian of an objective function $f : \mathbb{R}^n \to \mathbb{R}$ of the Lagrangian is of interest in the preprocessing step for the real optimization. Hence, sparsity is exploited to accelerate the Hessian accumulation using second-order adjoint model of AD as defined by Equation (1.11). However, estimation of the nonzero Hessian sparsity pattern also referred to as exact Hessian pattern (EHP) takes about 3400 seconds for a realistic input dimension $n = 16896276$. The overall computation time i.e. preprocessing and optimization is about 33100 seconds. Hence, the computational cost of estimating the exact Hessian pattern is roughly $\frac{1}{10} \approx \frac{3400}{33100}$ of the overall runtime , which is not really "negligibly" small.

In order to accelerate the sparsity pattern detection of Hessians in Section 3.3 we exploit the partial separability of the underlying function F. This results in a conservative overestimated version (CHP) of the pattern of target Hessians as introduced in Section 3.3.2. In the following we use also CHP to denote the respective algorithm. As already mentioned, estimating the exact Hessian pattern is of quadratic complexity. The reason for this lies in the propagation of second-order dependencies as set of index pair over the index domain X for every SAC variable v_j of F, that is, $sod(v_j) \subseteq X \times X$. Thereby, $sod(v_j)$ is computed as the cross product of first-order dependencies $fod(v_i)$ and the union of $sod(v_i)$ of their arguments $i \prec j$ in the case of nonlinear and linear operations, respectively.

CHP overcomes this problem by restricting the computationally expensive cross products and the unions of sods to the nonlinear components of F. Hence, CHP takes now only 11.7 seconds as opposed to 3400 seconds of EHP for inverse medium problem mentioned above. This means a runtime gain by a factor of roughly $290 \approx \frac{3400}{11.7}$. More interesting, the resulting Hessian pattern in both case are identical. This is because the nonlinear components of f consist of no multiplication. We note that the multiplication may produce overestimated Hessian entries. Hence, no change in the coloring performance in terms of achieved colors and runtime is expected.

Even more substantially, we observe even better coloring results using CHP in context of another objective function as shown in Figure 3.4 (a-c) with negligible loss in runtime, despite the fact that CHP is orders of magnitude faster than EHP as shown in (a). At this point we recapitulate that we observed similar behavior when coloring the variable Hessian pattern of the same problem, which is sparser than the exact (nonzero) one. We observe that the coloring underperforms by increase in sparsity for this test case. This observation seems to be surprising at first glance. However, a deeper look into, for instance, the sequential star coloring heuristic lead to the following. Therefore, we focus on the conservative and exact pattern of the target Hessian in Figure 3.5. The former consists of an overestimated (4×4) block in upper left corner. Consequently, the respective vertices can get the same colors in the adjacency graph of CHP. Henceforth, all most any other vertex can be colored by one of the used colors as opposed to its EHP counterpart. Hence, the conservative nonzero block seems to route the star coloring to a much better coloring, illustrates the importance of the structure of target Hessians. We believe this should be taken more into account in the (existing) coloring heuristics. It should also be mentioned that changing the ordering e.g. *smallest last* of vertices according their degrees (number of incident edges) in this case does not really help as the vertices are of almost the same degree.

Finally, Section 3.3.3 generalizes the exploitation of the partial separability yielding a recursive algorithm for Hessian pattern estimation denoted by RHP. First numerical results on an artificial example shows that RHP converges to CHP for sufficiently large recursion levels. Moreover, RHP at level one and CHP behave almost similarly in terms of runtime. We note that CHP is supposed to be obtained by RHP at recursion level one.

3.2 Quantitative Dependence Analysis

In the following we investigate methods for exploiting the sparsity of, in general, derivative tensor $\nabla^d F$. Existing compression techniques [CPR74] are based on the knowledge of the nonzeros. To achieve a better compression, we decompose the nonzeros into constants and variables; the former does not depend on the input values as opposed to the latter, that need to be computed at runtime. Thus, we consider

$$\nabla^d F = \nabla^d F_v + \nabla^d F_c \quad \text{with} \quad d \geq 1 \tag{3.1}$$

to be the sum of constant $\nabla^d F_c$ and variable $\nabla^d F_v$ parts. Henceforth, we use the terminology *constants* [*variables*] to refer to the former [latter]. Moreover, we provide dynamic algorithms for computing the constants along with the sparsity pattern of the variables and we prove their correctness.

Sparse derivative calculation, in general, consists of two main steps, namely sparsity detection and its exploitation in the process of derivative accumulation, where the former is often considered a preprocessor activity. A lot of work has already been done on this for Jacobian [TFE98, GPW08, NNH$^+$11] and Hessian [CM83, Wal08] computations as discussed in Section 3.2.4 and Section 3.2.5, respectively. In fact, sparsity detection can be performed either at runtime or by a compiler, where the former and latter are known as *dynamic* and *static* sparsity, respectively, the latter may result in a *conservative overestimation* of the target sparsity pattern as a consequence of existing control flow structures in F as discussed in [TFE98] for Jacobian matrices. However, our focus in the following is on dynamic sparsity and constant estimation that we refer to as *quantitative dependence analysis* (QDA) using operator overloading technique as a new variant of the function evaluation by propagating some *index domains* as done in the case of Jacobians and Hessians in ADOL-C. Obviously, the dynamic sparsity is valid only at the given point, which follows that any changes in control flow may result in recomputation of both sparsity pattern as well as constants. As an example throughout this chapter let us consider F in Example 3.1. A special variant of dependence analysis [BC04] is used to discover the dependencies of every output $y_{j=1,2}$ on every input $x_{i=1,2}$ represented as (y_j, x_i). This information yields the sparsity pattern of the Jacobian as the matrix of the first-order sensitivities $f'_{j,i}$ of F at $\mathbf{x} = (x_1, x_2)$ defined by Equation (1.7). Here, the dependencies (y_1, x_2) and (y_2, x_1) are of constant quantities -1 and 1, respectively.

Example 3.1. *As an example consider $F : \mathbb{R}^2 \to \mathbb{R}^2$ defined as follows.*

$$\begin{aligned} y_1 &= \sin(x_1) - x_2 \\ y_2 &= x_1 - x_2 \cdot x_2 \end{aligned} \tag{3.2}$$

The Jacobian of F is $\left(\begin{smallmatrix} \cos(x_1) & -1 \\ 1 & 2 \cdot x_2 \end{smallmatrix}\right)$. The entries $f'_{1,1}$ and $f'_{2,2}$ are variables, since they depend on the values of inputs x_1 and x_2, respectively. In the opposite $f'_{1,2}$ and $f'_{2,1}$ are constants.

Quantitative dependence analysis is concerned with the classification of the sensitivities of derivative tensors such as Jacobians or Hessians into *variable* (**v**), *constant nonzero* (**c**), and *zero* (**o**) types of sensitivities. Furthermore, *nonzeros* (**nz**) are considered the union over variables and constants. Thus, the outcomes of the quantitative dependence analysis of F are the sparsity pattern of the variables and the constants. Strictly speaking, QDA computes the sparsity pattern $P(\nabla^d F_v)$ of the variables $\nabla^d F_v$ and the constants $\nabla^d F_c$ of $\nabla^d F$ for a given derivative degree $d \geq 1$.

Assumption 3.1. *F of Equation (*1.1*) is canonical in the sense that no algebraic simplifications such as $\log(e^x) = x$, $\sqrt{x}^2 = x$, and $\frac{x^2-1}{x+1} = x - 1$ are possible.*

However, the target derivative tensor $\nabla^d F$ of F can be obtained in AD by the running a corresponding derivative code [Nau11, HP04] F^d. For d larger than one we talk about higher-order sensitivities, resulting from the application of the corresponding *higher-order derivative code*. Obviously, the sparsity as well as

constants of $\nabla^d F$ can be obtained by quantitative dependence analysis of the respective derivative codes. However, compared to F the size of such derivative codes grows, in general, exponentially with the derivative degree d. An example of such a code for $d = 1$ is given in listings 3.2. Moreover, higher ordered derivative codes usually consist of nested loops of depth growing linearly with d. Both facts complicate the QDA of such codes. To deal with this problem we propose a dynamic algorithm for performing QDA only on the original code F. Therefore, we refer to F^0 as the SAC of F at the current argument. Henceforth, whenever we talk about the example function, we mean F of Example 3.1 with the SAC given in Listing 3.1, where first and second assignments of F are decomposed into the SAC statements of lines 3-5 and 6-8, respectively. Thus, 0_F maps the independent inputs x1, x2 onto dependent outputs y1=v 2, y2=v4 involving the computation of intermediate values v1 and v3. Hence, $X = \{1, 2\}, Z = \{3, 5\}$, and $Y = \{4, 6\}$. Here we avoid explicitly independent assignments v1=x1 and v2=x2 for brevity as inputs are not overwritten.

Listing 3.1: 0_F

```
1 void 0_F( float x1, float x2, float& y1, float& y2) {
2   float v3, v4, v5, v6;
3   v3 = sin(x1);
4   v4 = v3 - x2;
5   y1 = v4;
6   v5 = x2 * x2;
7   v6 = x1 + v5;
8   y2 = v6;
9 }
```

3.2.1 Mathematical Background

We consider again the index set V defined by Equation (1.13) representing the indices of SAC variables the union of disjoint index sets $X = \{1, \cdots, n\}$, $Z = \{n+1, \ldots, n+p\}$, and $Y = \{n+p+1, \cdots, q\}$. For $k \in Z \cup Y$ we define

$$X_k = X(v_k) = \{i \in X : i \prec^* k, \} \quad \text{and} \quad \mathbf{x}_k = \mathbf{x}(v_k) = (x_i)_{i \in X_k} \tag{3.3}$$

as the *independent set* and *independent vector* of the SAC variable v_k, respectively. Given X

$$f : X \to I\!N, \quad A = (X, f) \quad \text{with} \quad d = \left|A\right| = \sum_{i \in X} f(i)$$

represents a multiset on X with the multiplicity function f and the cardinality d. Given $i \in X$, then $f(i)$ represents the number of the repetitions of the element i in the multiset A. The union $C = (X, h)$ of two multisets $A = (X, f)$ and $B = (X, g)$ over X is defined as

$$C = A \cup B, \quad \forall\, i \in X : \; h(i) = f(i) + g(i) \quad . \tag{3.4}$$

A is a submultiset of B denoted by

$$A \subseteq B \quad \text{if and only if} \quad \forall\, i \in X : f(i) \leq g(i) \quad .$$

A is a prober submultiset of B denoted by

$$A \subset B \quad \text{if and only if} \quad \forall\, i \in X : f(i) < g(i) \quad .$$

Moreover, we define

$$X^d = \{A = (X, f) : \left|A\right| = d\} \tag{3.5}$$

as the *domain* of $\nabla^d F$ consisting of all multisets of cardinality d on X. Obviously, $X^1 = \{< i >: i \in X\}$ and X are equivalent. Henceforth, we use the notation $\langle\cdot\rangle$ to denote multisets.

Example 3.2. *Given $X = 1, 2$ and multisets $A = \langle 1, 1, 1\rangle$, $B = \langle 1, 1, 2\rangle$, $C = \langle 1, 2, 2\rangle$, and $D = \langle 2, 2, 2\rangle$ of cardinality three. The union $A \cup B$ yields the multiset $\langle 1, 1, 1, 1, 1, 2\rangle$ of cardinality six on X. The set of all multisets on X with cardinality three is $X^3 = \{A, B, C, D\}$.*

Given the SAC variable v_k with $k \in V$ and the multiset $A = \langle i_1, \ldots, i_d\rangle \in X^d$ we refer to

$$c_{k,A} = c_{k,A}(\mathbf{x}) = \frac{\partial^d v_k}{\partial \mathbf{x}_A}(\mathbf{x}) = \frac{\partial^d v_k}{\partial x_{i_1} \ldots \partial x_{i_d}}(\mathbf{x}) \tag{3.6}$$

as the sensitivity of v_k with respect to the independents of A, which can be of variable, constant, or zero type as follows.

Definition 3.1. *The type of the partial derivative $c_{k,A}$ defined in Equation (*3.6*) of a SAC variable v_k, $k \in V$ with $\left|A\right| = d$ and $A \in X^d$ is*

1. *variable (**v**), if $c_{k,A}$ depends on $\mathbf{x}_k$ defined by Equation (*3.3*)*
2. *constant (**c**), if $c_{k,A}$ is nonzero not depending on $\mathbf{x}_k$, or*
3. *zero (**o**), otherwise.*

Hence, $c_{k,A}$ is considered a nonzero in the case of 1 *and* 2.

Furthermore, for $k \in V$ we define the *nonzero domain*

$$P^d(v_k) = \{A \in X^d : \exists \mathbf{x} \in D \text{ with } c_{k,A}(\mathbf{x}) \hat{=} \mathbf{nz}\} = P_v^d(v_k) \cup P_c^d(v_k) \quad ,$$

as the union of the variable and constant domains

$$P_v^d(v_k) = \{A \in P^d(v_k) : c_{k,A}(\mathbf{x}) \hat{=} \mathbf{v}\} \quad \text{and} \quad P_c^d(v_k) = \{A \in P^d(v_k) : c_{k,A} \hat{=} \mathbf{c}\} \quad ,$$

respectively, where the notation $c_{k,A} \hat{=} \mathtt{t}$ means that the partial derivative $c_{k,A}$ is of type $\mathtt{t} \in \{\mathbf{nz}, \mathbf{v}, \mathbf{c}, \mathbf{o}\}$. Given $i, j \in V$, and $1 \leq d_1, d_2$ with $d = d_1 + d_2$

$$P^{d_1}(v_i) \uplus P^{d_2}(v_j) = \{A = B \cup C : B \in P^{d_1}(v_i) \text{ and } C \in P^{d_2}(v_j)\}$$

denotes the *absorption* of two nonzero domains $P^{d_1}(v_i)$ and $P^{d_1}(v_j)$ with the union $B \cup C$ as defined by Equation (3.4). Obviously, the nonzero, variable, and constant domains of $\nabla^d F$ represent, in fact, the respective sparsity patterns. Henceforth, whenever we say that a variable v is nonzero, variable, and constant with respect to $A \in X^d$, we mean that $A \in P^d(v)$, $A \in P_v^d(v)$, and $A \in P_c^d(v)$, respectively.

3.2.2 Sparsity Pattern Estimation

In the following we consider

$$\Phi = \Phi_N \cup \{+, *\} \quad \text{with} \quad \Phi_N = \{\sin, \cos, \ldots, \text{pow}(u, v), \text{pow}(u, r)\} \quad ,$$

which is the minimal subset of the mathematical operations and intrinsic functions provided by the mathematic library `math.h` of the programming language C that we consider sufficient to illustrate the algorithmic behind QDA. However, a detailed view of provided operations and intrinsics is given in Table 3.1. We note that some of operations and intrinsics are explicitly missing in Φ either because they can be expressed using those of Φ or because they are integer operations. Here, the symbols v, u, w represent floating-point and n integer variables, whereas $c, r \in \mathbb{R}$ and $k \in I\!N$ denote constants. TSP estimation

Operations	Replaced by Expression	Ignored
$w = v$		No
$w = v * u$		No
$w = v + u$		No
$w = v * c = c * v$		No
$w = v + c = c + v$	$w = v$	No
$w = v - c$	$w = v$	No
$w = c - v$	$w = -1 * v$	No
$w = v/c$	$w = 1/c * v$	No
$w = c/v$	$w = c * \mathrm{pow}(v, -1)$	No
$w = v - u$	$w = v + (-1 * u)$	No
$w = v/u$	$w = v * \mathrm{pow}(u, -1)$	No
$w+ = v$	$w = w + v$	No
$w* = v$	$w = w * v$	No
$w/ = v$	$w = w * \mathrm{pow}(v, -1)$	No
$w- = v$	$w = w + (-1 * v)$	No
$\sin, \cos, \tan$		No
$\mathrm{asin}, \mathrm{acos}, \mathrm{atan}$		No
$\exp, \log, \log 10$		No
$w = \mathrm{pow}(v, u) = v^u$		No
$w = \mathrm{pow}(v, r) = v^r$		No
$w = \mathrm{atan}\,2(v, u)$	$w = \mathrm{atan}(v/u)$	No
$w = \mathrm{pow}(v, k)$	$w = \prod_{i=1}^{k} v$	No
$w = \mathrm{ldexp}(v, k)$	$w = v * \mathrm{pow}(2, k)$	No
$w = \mathrm{frexp}(v, \&n)$	$w = v/\mathrm{pow}(2, n)$	No
$w = \mathrm{modf}(v, \&n)$	$w = v - n$	No
$w = \mathrm{fmod}(v, u)$	$w = v - \mathrm{floor}(v, u) * u$	No
$w = \mathrm{fabs}(v) = \lvert v \rvert$	if $(v \geq 0)\, w = v$ else $w = -1 * v$	No
$w = \mathrm{ceil}(v) = \lceil v \rceil$		Yes
$w = \mathrm{floor}(v) = \lfloor v \rfloor$		Yes

Table 3.1: Operators and Intrinsics of `math.h` of the programming language C.

described in Algorithm 3.1 computes at runtime the *outgoing nonzero domain* $\mathbf{OutP}(\mathbf{F})$ of F on the corresponding SAC from the given *incoming nonzero domain* $\mathbf{InP}(\mathbf{F})$ defined as

$$\mathbf{OutP}(\mathbf{F}) = \begin{pmatrix} OutP^1(F) \\ \vdots \\ OutP^{d+1}(F) \end{pmatrix} \quad \text{and} \quad \mathbf{InP}(\mathbf{F}) = \begin{pmatrix} InP^1(F) = P_{id}(F) \\ InP^2(F) = \emptyset \\ \vdots \\ InP^{d+1}(F) = \emptyset \end{pmatrix} ,$$

where $P_{id}(F) = \{(x_i, P^1(x_i)) : i \in X\}$ and $OutP^l(F) = \{(y_{k-(n+p)}, P^l(v_k) : k \in Y\}$ with $P^1(x_i) = \{\langle i \rangle\}$ for $l = 1, \ldots, d+1$. The correctness of TSP is stated by Theorem 3.1. Obviously, the nonzero domain

$$P^d = P^d(\nabla^d F) = \bigcup_{j \in \{1,\ldots,m\}} P^d(y_j)$$

is the union of nonzero domains $P^d(y_j)$ of dependents y_j. At the same time, $P^d = P_v^d \cup P_c^d$ is the union of variable and constant domains

$$P_v^d = P(\nabla^d F_v) = \bigcup_{j \in \{1,\ldots,m\}} P_v^d(y_j) \quad \text{and} \quad P_c^d = P(\nabla^d F_c) = \bigcup_{j \in \{1,\ldots,m\}} P_c^d(y_j)$$

that results easily from Equation (3.1). The variable domain

$$P_v^d(y_j) = \{A \in P^d(y_j) : \exists\, B \in P^{d+1}(y_j) \text{ with } A \subset B\}$$

of y_j can be extracted from $P^d(y_j)$ by having $P^{d+1}(y_j)$ as a consequence of Lemma 3.1, which exploits the *proper submultiset property* between two multisets $A \in P^d(y_j)$ and $B \in P^{d+1}(y_j)$. Consequently, we can decompose the nonzero domain of $\nabla^d F$ into variables and constants by first computing the nonzero domains of dependents up to degree $d+1$ followed by extracting the variables.

Lemma 3.1. *Given $P^l(v_k)$ and $P^{l+1}(v_k)$ of the SAC variable v_k with $k \in Z \cup Y$ and $l \in \{1, \ldots, d\}$. The variable pattern $P_v^l(v_k)$ of v_k is computed as*

$$P_v^l(v_k) = \{A \in P^l(v_k) : \exists B \in P_k^{l+1} \textit{ with } A \subset B\} \quad .$$

Proof. We show that $\forall A \in P_v^l(v_k)$ there is a $B \in P_k^{l+1}$ such that $A \subset B$. Therefore, we consider $f(\mathbf{x}_k) = \frac{\partial^l v_k}{\partial \mathbf{x}_A}$ to be a function of independents $\mathbf{x}_k$. If $f(\mathbf{x}_k)$ is variable in some independent $i \in X_k$, then $\frac{\partial f(\mathbf{x}_k)}{\partial x_i}$ has to be nonzero. But, this means that

$$\frac{\partial f(\mathbf{x}_k)}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{\partial^l v_k}{\partial \mathbf{x}_A}\right] = \frac{\partial^{l+1} v_k}{\partial \mathbf{x}_B} \quad \text{with} \quad B = A \cup \{\langle i \rangle\} \quad \text{and} \quad A \subset B$$

represents a nonzero sensitivity, that is, $B \in P^{l+1}(v_k)$. □

Algorithm 3.1 (`TSP(d, SAC(F), InP(F), OutP(F))`: Tensor Sparsity Pattern Estimation)**.**

Require: derivative degree d, incoming nonzero domain $\mathbf{InP}(\mathbf{F})$.
Ensure: Outgoing nonzero domain $\mathbf{OutP}(\mathbf{F})$ of $\nabla^d F$.

1: **for** $i = 1$ **to** n **do**
2: $P^1(v_i) = P^1(x_i)$
3: **end for**
4: **for** $k = n+1$ **to** q **do**
5: **if** $v_k = v_i + v_j$; **then**
6: **for** $l = 1$ **to** $d+1$ **do**
7: $P^l(v_k) = \bigcup_{i \prec k} P^l(v_i)$
8: **end for**
9: **end if**
10: **if** $v_k = v_i \cdot v_j$ **then**

11: **for** $l = 1$ **to** $d + 1$ **do**
12: $P^l(v_k) = \bigcup_{i \prec k} P^l(v_i)$
13: **for** $l_1 = 1$ **to** $l - 1$ **do**
14: **for** $l_2 = 1$ **to** $l - 1$ **do**
15: **if** $l = l_1 + l_2$ **and** $P^{l_1}(v_i) \neq \emptyset$ **and** $P^{l_2}(v_j) \neq \emptyset$ **then**
16: $P^l(v_k) = P^l(v_k) \cup (P^{l_1}(v_i) \uplus P^{l_2}(v_j))$
17: **end if**
18: **end for**
19: **end for**
20: **end for**
21: **end if**
22: **if** $\varphi_k \in \Phi_N$ **then**
23: **for** $l = 1$ **to** $d + 1$ **do**
24: $A = \bigcup_{i \prec k} X_i$
25: $P^l(v_k) = A^l$
26: **end for**
27: **end if**
28: **end for**
29: **for all** $k \in Y$ **do**
30: **for** $l = 1$ **to** $d + 1$ **do**
31: $OutP^l = OutP^l \cup (y_{k-(n+p)},\ P^l(v_k))$
32: **end for**
33: **end for**

Theorem 3.1. *Algorithm 3.1 computes the correct nonzero domain $P^l(v_k)$ of the SAC variable v_k with $k \in V$ for $l = 1, \ldots, d + 1$.*

Proof. We consider $A \in X^l$ with $1 \leq l \leq d + 1$.

- $\mathbf{InP}(\mathbf{F})$ is correct, since the first partial derivative of every independent variable x_k with $k \in X$ is only with respect to itself nonzero otherwise zero.
- Lines 1-3 initialize the nonzero domains of the independent SAC variables to those of inputs X.
- Lines 5-9: line 7 follows from the addition rule for higher partial derivatives yielding

$$\frac{\partial^l v_k}{\partial \mathbf{x}_A} = \frac{\partial^l v_i}{\partial \mathbf{x}_A} + \frac{\partial^l v_j}{\partial \mathbf{x}_A} \quad . \tag{3.7}$$

  Hence, $\frac{\partial^l v_k}{\partial \mathbf{x}_A}$ is nonzero, if and only if either $\frac{\partial^l v_i}{\partial \mathbf{x}_A}$ or $\frac{\partial^l v_j}{\partial \mathbf{x}_A}$ is nonzero, that is, $A \in P^l(v_i) \cup P^l(v_j)$.
- Lines 10-21: Equations in lines 12 and 16 follow from the Leibniz product rule for higher partial derivatives yielding

$$\frac{\partial^l v_k}{\partial \mathbf{x}_A} = \sum_{\forall\, B \subseteq A} \frac{\partial^{l_1} v_i}{\partial \mathbf{x}_B} \cdot \frac{\partial^{l_2} v_j}{\partial \mathbf{x}_C} \quad \text{with} \quad C = A - B, \quad l_1 = \left|B\right|, \quad \text{and} \quad l_2 = \left|C\right| . \tag{3.8}$$

  Hence, $\frac{\partial^l v_k}{\partial \mathbf{x}_A}$ is nonzero if and only if either

  1. $\frac{\partial^l v_i}{\partial \mathbf{x}_A} \neq 0$ i.e. $A \in P^l(v_i)$ for $B = A, C = \emptyset$ or,

2. $\frac{\partial^l v_j}{\partial \mathbf{x}_A} \neq 0$ i.e. $A \in P^l(v_j)$ for $B = \emptyset, C = A$ or,
3. $\frac{\partial^{l_1} v_i}{\partial \mathbf{x}_B} \neq 0$ and $\frac{\partial^{l_2} v_j}{\partial \mathbf{x}_C} \neq 0$ i.e. $B \in P^{l_1}(v_i)$ and $C \in P^{l_2}(v_j)$ otherwise.

- Lines 22-27 : Equation in line 25 follows from the differentiation rule for pure nonlinear functions of Φ_N.
- Lines 29-33 : each dependent variable $y_{k-(n+p)}$ with $k \in Y$ contributes in line 31 its nonzero domains $P^l(v_k)$ to $OutP^l(F)$.

□

It can be shown by induction that the worst case complexity of the tensor sparsity pattern estimation described in Algorithm 3.1 for a given degree $d \geq 1$ is as

$$OPS(TSP) \in O(d \cdot \hat{n}^{d+1}) \quad , \tag{3.9}$$

where

$$\hat{n} = \max_{i \in V} |P^1(v_i)| \tag{3.10}$$

denotes the maximum number of elements of *first-order dependencies* $P^1(v_i)$ over all SAC variables v_i of F with $i \in V$. Moreover, $OPS(F)$ denotes the number of floating point operations in the SAC of F.

The main contribution to this complexity is made by nonlinear operations of lines 22 and 10. Let us consider first the multiplication in the latter. Furthermore, let us assume to be interested in $d = 1$. The union in line 12 can be performed in $O(\hat{n})$, which follows immediately from Equation (3.10). Hence, the absorption of $P^1(v_i)$ and $P^1(v_j)$ for $l_1 = 1$ and $l_2 = 1$ of the arguments of v_k in line 16 along with the union of the result yields $|P^2(v_k)| \in O(\hat{n}^2)$ at the same quadratic computational cost.

Now let us assume $d = 2$. The absorptions of $P^{l_1}(v_i)$ and $P^{l_2}(v_j)$ for $l_1 = 1, l_2 = 2$, and $l_1 = 2, l_2 = 1$ result in $P^3(v_k)$, where $|P^3(v_k)| \in O(\hat{n}^3)$. The cardinality and the complexity of the absorption results from the cardinality of the arguments as shown above for $d = 1$.

Likewise, cardinality and complexity are achieved for $P^l(v_k)$ of line 25. From differentiation rule of pure nonlinear functions it follows that $P^l(v_k) = A^l$ for $l = 1, \ldots, d+1$. Note that $|A| \in O(\hat{n})$ and A^l denote the set of all multiset of cardinality l over index set A as defined by Equation (3.5). Hence, it can be shown that $|P^l(v_k)| \in O(\hat{n}^l)$.

Finally, the factor d to the complexity results from the fact that the absorption and union operations of line 16 are performed for d different combinations of l_1 and l_2 such that $d + 1 = l_1 + l_2$.

3.2.3 Computation of Constant Partial Derivatives

Algorithm 3.2 describes the computation of the constant partial derivatives of $\nabla^d F$ separated from the pattern computation for simplicity on the SAC of F. Here, it assumes that at the time of computation of constants C_k^l for $k \in V$ the corresponding nonzero domains $P^l(v_k)$ and $P^{l+1}(v_k)$ for $l = 1, \ldots, d$ are given. These are needed to separate variables from constants. Thus, constants

$$C^d(v_k) = \{(A, c_{k,A}) : A \in X^d \text{ and } c_{k,A} \hat{=} \mathbf{c}\}$$

of each SAC variable v_k consists of tuples $(A, c_{k,A})$ with the constant sensitivities $c_{k,A}$. Furthermore, we define the *absorption*

$$C^{l_1}(v_i) \uplus C^{l_2}(v_j) = \{(A, a \cdot b) : (B, a) \in C^{l_1}(v_i) \text{ and } (C, b) \in C^{l_2}(v_j)\}$$

of two constants $C^{l_1}(v_i)$ and $C^{l_2}(v_j)$ with $i, j \in V$ as used in line 21 of Algorithm 3.2. Moreover, we assume that the union of $C^l_{v_i}$ and $C^l_{v_j}$ used in lines 12 and 21 has the property that

$$\forall A \in X^l \text{ if } (A, a) \in C^l_{v_i} \text{ and } (A, b) \in C^l_{v_j} \quad \Rightarrow \quad (A, a+b) \in C^l_{v_i} \cup C^l_{v_j} \quad .$$

Thus, Algorithm 3.2 computes at runtime the *outgoing constants* $\mathbf{OutC}(\mathbf{F})$ of F on the corresponding SAC from given *incoming constants* $\mathbf{InC}(\mathbf{F})$ defined as

$$\mathbf{OutC}(\mathbf{F}) = \begin{pmatrix} OutC^1(F) \\ \vdots \\ OutC^d(F) \end{pmatrix} \quad \text{and} \quad \mathbf{InC}(\mathbf{F}) = \begin{pmatrix} InC^1(F) = C_{id}(F) \\ InC^2(F) = \emptyset \\ \vdots \\ InC^d(F) = \emptyset \end{pmatrix} ,$$

where $C_{id}(F) = \{(x_i, C^1(x_i)) : i \in X\}$ and $OutC^l(F) = \{\big(y_{k-(n+p)}, C^l(v_k)\big) : k \in Y\}$ with $C^1(x_i) = \{(\langle i \rangle, 1)\}$ for $l = 1 \ldots, d$. Obviously and as shown in line 41 the constants of $\nabla^d F$ results from the union

$$C^d(F) = \bigcup_{j \in \{1,\ldots,m\}} C^d(y_j)$$

of constants $C^d(y_j)$ of the dependents y_j. Theorem 3.2 proves the correctness of the constant computation by Algorithm 3.2. We note that the most important operation in terms of constant generation as well as computational cost is the multiplication as shown in lines 15-27. Thereby, constants generated by the multiplication may get destroyed later on as shown in lines 12, 25, and 30. For instance, consider the computation of constants of the following statements for $d = 2$.

$$\begin{aligned} v &= x_1 \cdot x_2; \quad //\frac{\partial^2 v}{\partial x_1 \partial x_2} = 1 \mathrel{\hat{=}} \mathbf{c} \\ u &= \sin(v); \quad //\frac{\partial^2 u}{\partial x_1 \partial x_2} = \cos(v) - x_2 \cdot \sin(v) \cdot x_1 \mathrel{\hat{=}} \mathbf{v} \\ w &= v \cdot x_2; \quad //\frac{\partial^2 w}{\partial x_1 \partial x_2} = 2 \cdot x_2 \mathrel{\hat{=}} \mathbf{v} \\ z &= v + w; \quad //\frac{\partial^2 z}{\partial x_1 \partial x_2} = 1 + 2 \cdot x_2 \mathrel{\hat{=}} \mathbf{v} \end{aligned}$$

As one can easily see the second-order sensitivity $\frac{\partial^2 v}{\partial x_1 \partial x_2}$ of v with respect to $\langle 1, 2 \rangle$ is constant. Hence we get $C^2(v) = \{(\langle 1, 2 \rangle, 1.)\}$. However, none of the succeeding three statements u, w, and z are constant but variable with respect to A. The reason is that the constants of v gets destroyed in u by nonlinear intrinsic $\sin$, in w by multiplying v with x_2 and finally in z by adding v and w.

Now, let $d = 3$. Hence $\frac{\partial^3 w}{\partial x_1 \partial x_2 \partial x_2} = 2$ as $w = x_1 \cdot x_2 \cdot x_2$, which is in fact $w = x_1 \cdot \mathrm{pow}(x_2, 2.)$. However, we get $C^3(w) = \{(\langle 1, 2, 2 \rangle, 1.)\}$ by the multiplication rule in line 21 for $C^1(x_2) = \{(\langle 2 \rangle, 1.)\}$ and $C^2(v) = \{(\langle 1, 2 \rangle, 1.)\}$. Finally, we get $C^3(w) = \{(\langle 1, 2, 2 \rangle, 2)\}$ by applying the power rule in line 38 for $l = 2$, $f(1) = 1$, and $f(2) = 2$.

In the following we use the notations $A \in P^l$ [$A \notin P^l$] for a given multiset A with $|A| < l$ and $l \geq 2$ to denote that

$$\exists B \in P^l : A \subset B \quad [\nexists B \in P^l : A \subset B] \quad .$$

Algorithm 3.2 ($\texttt{TCE}(\texttt{d}, \texttt{SAC}(\texttt{F}), \texttt{InC}(\texttt{F}), \texttt{OutC}(\texttt{F}))$: Tensor Constant Estimation**).**

Require: derivative degree d, incoming constants $\mathbf{InC}(\mathbf{F})$.
Ensure: Outgoing constant domain $\mathbf{OutC}(\mathbf{F})$ of $\nabla^d F$.

1: **for** $i = 1$ **to** n **do**
2: $C^1(v_i) = C^(x_i)$
3: **end for**
4: **for** $k = n + 1$ **to** q **do**
5: **if** $v_k = c \cdot v_j$ **then**
6: **for** $l = 1$ **to** d **do**
7: $C^l(v_k) = \{(A, c_{k,A}) : c_{k,A} = c \cdot c_{j,A}\}$
8: **end for**
9: **end if**
10: **if** $v_k = v_i + v_j$ **then**
11: **for** $l = 1$ **to** d **do**
12: $C^l(v_k) = C^l(v_i) \cup C^l(v_j) - \{(A, *) : A \in P_k^{l+1}\}$
13: **end for**
14: **end if**
15: **if** $v_k = v_i \cdot v_j$ **then**
16: **for** $l = 1$ **to** d **do**
17: $C^l(v_k) = \emptyset$
18: **for** $l_1 = 1$ **to** $l - 1$ **do**
19: **for** $l_2 = 1$ **to** $l - 1$ **do**
20: **if** $l = l_1 + l_2$ **and** $C^{l_1}(v_i) \neq \emptyset$ **and** $C^{l_2}(v_j) \neq \emptyset$ **then**
21: $C^l(v_k) = C^l(v_k) \cup (C^{l_1}(v_i) \uplus C^{l_2}(v_j))$
22: **end if**
23: **end for**
24: **end for**
25: $C^l(v_k) = C^l(v_k) - \{(A, *) : A \in P_k^{l+1}\}$
26: **end for**
27: **end if**
28: **if** $\varphi_k \in \Phi_N$ **then**
29: **for** $l = 1$ **to** d **do**
30: $C^l(v_k) = \emptyset$
31: **end for**
32: **end if**
33: **end for**
34: **for all** $k \in Y$ **do**
35: **for** $l = 1$ **to** d **do**
36: **for all** $(A, c) \in C^l(v_k)$ **do**
37: **for all** $i \in A$ with $f(i) > 1$ **do**
38: $c = c \cdot \prod_{j=1}^{\min(l, f(i))} (f(i) - (j-1))!$
39: **end for**
40: **end for**
41: $OutC^l = OutC^l \cup \{(y_{k-(n+p)},\ C^l(v_k)\}$
42: **end for**
43: **end for**

Theorem 3.2. *Algorithm 3.2 computes the correct constants C_k^l of the SAC variable v_k with $k \in V$ for $l = 1, \ldots, d$.*

Proof. Let us consider $A(X, f)$ with $\left|A\right| = l$ and $1 \leq l \leq d$.

- $\mathbf{InC}(\mathbf{F})$ is correct, since the first partial derivative of every independent variable x_k with $k \in X$ is with respect to itself constant one otherwise zero.
- Lines 1-3 initialize constants SAC independent variables v_k with $k \in X$ to those of inputs X.
- Lines 5-9 : Equation in line 12 follows from the application of Leibniz product defined by Equation (3.8) for higher partial derivatives with $v_i = c$ yielding $\frac{\partial^l v_k}{\partial \mathbf{x}_A} = c \cdot \frac{\partial^l v_j}{\partial \mathbf{x}_A}$. Hence,

$$(A, c_{j,A}) \in C^l(v_j) \quad \Rightarrow \quad (A, c \cdot c_{j,A}) \in C^l(v_k) \quad .$$

- Lines 10–14 : W.l.o.g. we assume that $A \in P^l(v_i)$ and $A \in P^l(v_j)$. Hence, Equation in line 12 follows from addition rule for higher partial derivatives. Hence, $\frac{\partial^l v_k}{\partial \mathbf{x}_A}$ of Equation (3.7) is constant if $A \notin P^{l+1}$, that is, if $(A, c_{i,A}) \in C^l(v_i)$ and $(A, c_{j,A}) \in C^l(v_j)$.
- Lines 15-21 : Equations in lines 21 and 25 follows from the facts that
  1. if $A \in P^{l+1}(v_k)$ then v_k is not constant with respect to A
  2. otherwise $c_{k,A}$ of Equation (3.8) is constant, that is, $(A, c_{k.A}) \in C^l(v_k)$, if and only if

$$\forall B \subset A : (B, c_{i,B}) \in C^{|B|}(v_i) \text{ and } (D, c_{j,D}) \in C^{|D|}(v_j) \quad ,$$

  where $D = A - B$, $B \in P^{|B|}(v_i)$, and $D \in P^{|D|}(v_j)$.
- Lines 28-32 : Equation in line 30 follow from the fact that nonlinearity destroys constants.
- Lines 34-43 : dependent variables $y_{k-(n+p)}$ with $k \in Y$ contribute in line 41 their constants C_k^l to $OutC^l(F)$. However, previously in line 38 the corresponding constants are multiplied by the respective factor according to the power rule to yield right constant values in case of element repetitions.

□

We note here that the computational complexity of tensor constant estimation TCE described in Algorithm 3.2 is bounded by that of TSP defined by Equation (3.9) as

$$OPS(TCE) \leq OPS(TSP) \quad .$$

Note that the respective SAC operation in TSP is supposed to be performed prior to that of TCP enabling the separation of the variables from the constants. One can easily figure out that $|C^l(v_k)| \leq |P^l(v_k)|$ for all $i \in V$ for $l = 1, \ldots, d$. Hence, the absorptions and unions can be performed in $O(\hat{n}^l)$ in worst case. However, the computation of constants along with the variable pattern results in more computational effort even though the complexity class remains unchanged. Our experimental results in context of sparse Jacobian and Hessian computations will show that the gain from constant exploitation depends very much on the problem F and the coloring heuristics of use as well. The Fact is that even variable pattern estimation by TSP is a complexity class higher than the estimation of the nonzero pattern. Hence, in the case of denser Jacobians or Hessians the runtime overhead of computing constants might be acceptable.

Example 3.3. *In the following we compute the nonzero domains and constants of the Jacobian* $\nabla F(x_1, x_2)$ *and Hessian* $\nabla^2 F(x_1, x_2)$ *on the SAC of our example function* F *by Equation (*3.2*). The SAC variables v2 and v4 represent respectively the dependents* y_1 *and* y_2*, as shown in lines* 5 *and* 8 *of Listing 3.1. We set* $\mathtt{P}^1(x1) = \{\langle 1\rangle\}, \mathtt{P}^1(x2) = \{\langle 2\rangle\}, \mathtt{C}^1(x1) = \{(\langle 1\rangle, 1)\}$*, and* $\mathtt{C}^1(x2) = \{(\langle 2\rangle, 2)\}$ *and get the following.*

$$\mathtt{InP}^1(\mathtt{F}) = \{(x1, \mathtt{P}^1(x1)), (x2, \mathtt{P}^1(x2))\}; \quad \mathtt{InP}^2(\mathtt{F}) = \emptyset; \quad \mathtt{InP}^3(\mathtt{F}) = \emptyset;$$
$$\mathtt{InC}^1(\mathtt{F}) = \{(x1, \mathtt{C}^1(x1)), (x2, \mathtt{C}^1(x2))\}; \quad \mathtt{InC}^2(\mathtt{F}) = \emptyset;$$

1. *v1 = sin(x1)*

$$\mathtt{P}^1_1 = \{\langle 1\rangle\}; \quad \mathtt{P}^2_1 = \{\langle 1, 1\rangle\}; \quad \mathtt{P}^3_1 = \{\langle 1, 1, 1\rangle\};$$
$$\mathtt{C}^1_1 = \emptyset; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},1} = \mathtt{P}^1_1;$$
$$\mathtt{C}^1_1 = \emptyset; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},1} = \mathtt{P}^2_1;$$

2. *v2 = v1 − x2*

$$\mathtt{P}^1_2 = \{\langle 1\rangle, \langle 2\rangle\}; \quad \mathtt{P}^2_2 = \{\langle 1, 1\rangle\}; \quad \mathtt{P}^3_2 = \{\langle 1, 1, 1\rangle\};$$
$$\mathtt{C}^1_2 = \{(\langle 2\rangle, -1)\}; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},2} = \{\langle 1\rangle\};$$
$$\mathtt{C}^2_2 = \emptyset; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},2} = \mathtt{P}^2_2;$$

3. *y1 = v2*

$$\mathtt{P}^1(y1) = \mathtt{P}^1_2; \quad \mathtt{P}^2(y1) = \mathtt{P}^2_2; \quad \mathtt{P}^3(y1) = \mathtt{P}^3_2;$$
$$\mathtt{C}^1(y1) = \{(\langle 2\rangle, -1.)\}; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v}}(y1) = \{\langle 1\rangle\};$$
$$\mathtt{C}^2(y1) = \emptyset; \quad \rightarrow \quad \mathtt{P}^2_{\mathtt{v}}(y1) = \mathtt{P}^2(y1);$$

4. *v3 = x2 ∗ x2*

$$\mathtt{P}^1_3 = \{\langle 2\rangle\}; \quad \mathtt{P}^2_3 = \{\langle 2, 2\rangle\}; \quad \mathtt{P}^3_3 = \emptyset;$$
$$\mathtt{C}^1_3 = \emptyset; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},3} = \mathtt{P}^1_3;$$
$$\mathtt{C}^2_3 = \{(\langle 2, 2\rangle, 1.)\}; \quad \rightarrow \quad \mathtt{P}^2_{\mathtt{v},3} = \emptyset;$$

5. *v4 = x1 + v3*

$$\mathtt{P}^1_4 = \{\langle 1\rangle, \langle 2\rangle\}; \quad \mathtt{P}^2_4 = \{\langle 2, 2\rangle\}; \quad \mathtt{P}^3_4 = \emptyset;$$
$$\mathtt{C}^1_4 = \{(\langle 1\rangle, 1.)\}; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v},4} = \{\langle 2\rangle\};$$
$$\mathtt{C}^2_4 = \{(\langle 2, 2\rangle, 1.)\}; \quad \rightarrow \quad \mathtt{P}^2_{\mathtt{v},4} = \emptyset;$$

6. *y2 = v4*

$$\mathtt{P}^1(y2) = \mathtt{P}^1_4; \quad \mathtt{P}^2(y2) = \mathtt{P}^2_4; \quad \mathtt{P}^3(y2) = \emptyset;$$
$$\mathtt{C}^1(y2) = \{(\langle 1\rangle, 1.)\}; \quad \rightarrow \quad \mathtt{P}^1_{\mathtt{v}}(y2) = \{\langle 2\rangle\};$$
$$\mathtt{C}^2(y2) = \{(\langle 2, 2\rangle, 2.)\}; \quad \rightarrow \quad \mathtt{P}^2_{\mathtt{v}}(y2) = \emptyset;$$

Hence, we obtain the following:

$$\begin{aligned}
\texttt{OutP}^1(\texttt{F}) &= \{(y1, \texttt{P}^1(y1)), (y2, \texttt{P}^1(y2))\};\\
\texttt{OutP}^2(\texttt{F}) &= \{(y1, \texttt{P}^2(y2)), (y2, \texttt{P}^2(y2))\};\\
\texttt{OutP}^3(\texttt{F}) &= \{(y1, \texttt{P}^3(y1))\};\\
\texttt{OutC}^1(\texttt{F}) &= \{(y1, \texttt{C}^1(y1)), (y2, \texttt{C}^1(y2))\};\\
\texttt{OutC}^2(\texttt{F}) &= \{(y2, \texttt{C}^2(y2))\};
\end{aligned}$$

From $\texttt{P}^1(y1)$ *and* $\texttt{P}^1(y2)$ *follows that the Jacobian* $\nabla F(x_1, x_2)$ *is entirely nonzero. From* $\langle 1\rangle \in \texttt{P}_V^1(y1)$ *and* $\langle 2\rangle \in \texttt{P}_V^1(y2)$ *follows that* $f'_{1,1}$ *and* $f'_{2,2}$ *are of variable type. Likewise, the constance of* $f'_{1,2} = -1$ *and* $f'_{2,1} = 1$ *follows from* $(\langle 2\rangle, -1) \in \texttt{C}_V^1(y1)$ *and* $(\langle 1\rangle, 1.) \in \texttt{C}_V^2(y1)$*, respectively. In the same way, from* $\langle 1, 1\rangle \in \texttt{P}_V^2(y1)$ *and* $(\langle 2, 2\rangle, 2.) \in \texttt{C}^2(y2)$ *it follows that the entries* $f''_{1,1,1}$ *and* $f''_{2,2,2}$ *of the Hessian* $\nabla^2 F(x_1, x_2)$ *defined by Equation (*1.12*) are variable and constant, respectively.*

3.2.4 Case Study I : Sparse Jacobian Computation

The [transposed] Jacobian of F can be computed using TLVM [ADVM] of F as defined in Equation (1.8) [(1.9)] yielding compressed [transposed] Jacobian

$$\mathbb{R}^{m\times p} \ni \tilde{B} = \nabla F(\mathbf{x}) \cdot \tilde{S} \in \mathbb{R}^{n\times p} \quad \left[\mathbb{R}^{n\times q} \ni \bar{B} = \nabla F(\mathbf{x})^T \cdot \bar{S} \in \mathbb{R}^{m\times q}\right] \tag{3.11}$$

as the result of p [q] evaluations of the respective TLM [ADJM] defined by Equation (1.3) [(1.5)]. The *seed matrix* $\tilde{S} \in \mathbb{R}^{n\times p}$ [$\bar{S} \in \mathbb{R}^{m\times q}$] is the result of partitioning the Jacobian into p [q] groups of *structurally orthogonal* columns [rows] [CPR74]. Two columns [rows] i and j are structurally orthogonal, if there is no row [column] k with $f'_{k,i} \neq 0$ and $f'_{k,j} \neq 0$ [$f'_{i,k} \neq 0$ and $f'_{j,k} \neq 0$]. Thus, an entry (i, k) of $\tilde{S}$ [$\bar{S}$] is one if the ith column [row] of the Jacobian belongs to the group k, and zero otherwise. The combinatorial problem is to find a minimal p [q], which can be stated as coloring problems [GMP05] that is known to be NP-complete on various graph representations $G(P)$ with

$$P \equiv (p_{j,i})_{i=1,\ldots,n}^{j=1,\ldots,m} \quad \text{with} \quad p_{j,i} \in \{0, 1\}$$

denoting the sparsity pattern of ∇F. In the following and w.l.o.g. we denote the bipartite graph by $G(P)$. Thus, we obtain $\tilde{S}$ [$\bar{S}$] by application of the *partial distance-2 coloring* algorithm as implemented in the graph coloring package ColPack[1] to the column [row] vertices of $G(P)$. Two vertices can get the same color, if they are not connected via a path of length two, otherwise they get different colors. Finally, we recover the nonzero entries of ∇F from $\tilde{B}$ [$\bar{B}$] using a simple substitution procedure as described in Algorithm 3.3.

Procedure 3.1. *The entire process of sparse Jacobian computation (SJC1) is as follows:*

S1. Evaluation of F *at the given point yields* P*,*

S2. Coloring column [row] vertices of $G(P)$ *yields* $\tilde{S}$ *[*$\bar{S}$*],*

S3. TLVM [ADVM] with seed matrix $\tilde{S}$ *[*$\bar{S}$*] yields* $\tilde{B}$ *[*$\bar{B}$*], and*

*S4. Recovery using Algorithm 3.3 yields the solution of Equation (*3.11*) for unknown entries of* ∇F*.*

[1] http://www.cscapes.org/coloringpage/

Constant Exploitation

To achieve a better compression, we consider in the following the Jacobian $\nabla F = \nabla F_v + \nabla F_c$ as the sum of its variable and constant entries as

$$\nabla F_v \equiv (v_{j,i})_{i=1,\ldots,n}^{j=1,\ldots,m} \quad \text{and} \quad \nabla F_c \equiv (c_{j,i})_{i=1,\ldots,n}^{j=1,\ldots,m} \quad .$$

We compute the sparsity pattern $P_v = P(\nabla F_v)$ resp. ∇F_c by applying Algorithm 3.1 resp. Algorithm 3.2 for $d = 1$ and obtain

$$\tilde{S}_v \in \mathbb{R}^{n \times p_v} \quad \text{and} \quad \bar{S}_v \in \mathbb{R}^{m \times q_v}$$

by *distance-2 coloring* of the column and row vertices of $G(\nabla F_v)$, respectively. Thus, application of TLVM resp. ADVM yields the compressed Jacobian $\tilde{D} \in \mathbb{R}^{m \times p_v}$ resp. $\bar{D} \in \mathbb{R}^{n \times q_v}$ as

$$\tilde{D} = \nabla F \cdot \tilde{S}_v = \nabla F_v \cdot \tilde{S}_v + \nabla F_c \cdot \tilde{S}_v \quad \text{resp.} \quad \bar{D} = \nabla F^T \cdot \bar{S}_v = \nabla F_v^T \cdot \bar{S}_v + \nabla F_c^T \cdot \bar{S}_v \quad .$$

Having $\tilde{D}$ resp. $\bar{D}$ we obtain ∇F_v by solving the linear system

$$\tilde{D} - F_c' \cdot \tilde{S}_v = F_v' \cdot \tilde{S}_v \quad \text{resp.} \quad \bar{D} - \nabla F_c^T \cdot \bar{S}_v = \nabla F_v^T \cdot \bar{S}_v \tag{3.12}$$

using Algorithm 3.4 to recover column resp. row entries of ∇F_v.

Procedure 3.2. *The entire process of sparse Jacobian computation with constant exploitation (SJC2) is as follows:*

S1. Evaluation of F at the given point yields P_v and ∇F_c,

S2. Coloring column [row] vertices of $G(P_v)$ yields $\tilde{S}_v$ [$\bar{S}_v$],

S3. TLVM [ADVM] with seed matrix $\tilde{S}_v$ [$\bar{S}_v$] yields $\tilde{D}$ [$\bar{D}$], and

S4. Recovery using Algorithm 3.4 yields the solution of Equation (3.12) for unknown entries of ∇F_v.

Listing 3.2: 1_F

```
1 void 1_F(float x1, float& d_x1,
2 float x2, float& d_x2,
3 float& y1, float& d_y1,
4 float& y2, float& d_y2, int p) {
5
6   float v1, v2, v3, v4;
7   float d_v1[p], d_v2[p], d_v3[p], d_v4[p];
8
9   int i;
10  v1 = sin(x1);
11  v2 = v1 - x2;
12  y1 = v2;
13  v3 = x2 * x2;
14  v4 = x1 + v3;
15  y2 = v4;
16  for (i = 0; i < p; ++i) {
17    d_v1[i] = d_x1[i]*cos(x1);
```

```
18      d_v2[i] = d_v1[i] - d_x2[i];
19      d_y1[i] = d_v2[i];
20      d_v3[i] = d_x2[i]*x2 + x2*d_x2[i];
21      d_v4[i] = d_x1[i] + d_v3[i];
22      d_y2[i] = d_v4[i];
23    }
24 }
```

1_F in Listing 3.2 represents the corresponding TLVM code of example function F of Example 3.1. No compression is possible for ∇F, since there are no structurally orthogonal columns. Hence, we set $S = I_2$ to get the full Jacobian entries

$$\tilde{B}_{1,1} = \text{d\_y1[0]} = \cos(x_1), \quad \tilde{B}_{1,2} = \text{d\_y1[1]} = -1, \quad \tilde{B}_{2,1} = \text{d\_y2[0]} = 1, \quad \tilde{B}_{2,2} = \text{d\_y2[1]} = 2 \cdot x_2$$

by calling 1_F with *seeding*

$$\text{d\_x1[0]} = 1, \quad \text{d\_x1[1]} = 0, \quad \text{d\_x2[0]} = 0, \quad \text{d\_x2[1]} = 1 \quad .$$

However, columns one and two of ∇F_v are structurally orthogonal yielding the seed matrix $\tilde{S}_v = \binom{1}{1}$. Now, we get the compressed Jacobian entries

$$\tilde{D}_{1,1} = \cos(x_1) - 1 \quad \text{and} \quad \tilde{D}_{2,1} = 1 - 2 \cdot x2$$

by calling 1_F with *seeding*

$$\text{d\_x1[0]} = 1 \quad \text{and} \quad \text{d\_x2[0]} = 1 \quad .$$

To recover $v_{1,1}$ from $\tilde{D}$ we subtract the constant $c_{1,2} = -1$ from the compressed entry $\tilde{D}_{1,1}$ to get $v_{1,2} = \cos(x_1)$. Similarly we subtract the constant $c_{2,1} = 1$ from $\tilde{D}_{2,1}$ and yield $v_{2,2} = -2 \cdot x_2$ which solves Equation (3.12), where

$$\nabla F_v = \begin{pmatrix} \cos(x1) & 0 \\ 0 & -2 \cdot x2 \end{pmatrix}, \quad \nabla F_c = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad \text{and} \quad \tilde{D} = \begin{pmatrix} \cos(x1) - 1 \\ 1 - 2 \cdot x2 \end{pmatrix} \quad .$$

The left column of Figure 3.1 illustrates graphically SJC2 representing the entire process of Jacobian accumulation with constant exploitation described in Procedure 3.2. However, in the following we assume w.l.o.g to be interested in Jacobians of F at inputs $\mathbf{x}_i$ in $I \subseteq D$ with $i \geq 1$, where $\mathbf{x}_1$ denotes the starting point under Assumption 2.1. We note again that any changes in control flow results conservatively in recomputation of both sparsity pattern as well as constants. Thus, first Jacobian $\nabla F(\mathbf{x}_1)$ is accumulated at starting point $\mathbf{x}_1$ by performing the steps S1, S2, S3, and S4, whereas all others result from steps S3 and S4 as P_v as well as ∇F_c remain unchanged in I. Here, we assume that the termination of step S4 is followed by a jump to evaluation process, whenever the Jacobians at another point is of interest, otherwise, S4 is supposed to finalize SJC2.

Another way to obtain constants of the Jacobian is illustrated in the right column of Figure 3.1. Here, we avoid the computation overhead of constants on the SAC of F as follows. At the starting point $\mathbf{x}_1$ we obtain the Jacobian by SJC1 described in Procedure 3.1 with only difference that we also estimate P_v along with P. Obviously, we get the constant pattern as $P_c = P - P_v$. After termination of the recovery in step S4 yielding $\nabla F(\mathbf{x}_1)$ and since we have P_c we obtain easily ∇F_c from ∇F. Furthermore, we obtain in step S2 both seed matrices $\tilde{S}$ resp. $\tilde{S}_v$ [$\bar{S}$ resp. $\bar{S}_v$] by distance-2 coloring of column [row] vertices of $G(P)$ resp. $G(P_v)$ such that in the following iterations we proceed to accumulate the target Jacobians just by performing the steps S3 and S4 of SJC2 shown in the left column of the same figure.

Procedure 3.3. *The alternative way to compute and exploit the Jacobian constants (SJC3) is as follows:*

- *At first iteration i.e. for $i = 1$ we perform an extended version of SJC1 as follows:*
  - *S1. Evaluation of F at the given point yields P and P_v,*
  - *S2. Coloring column [row] vertices of $G(P)$ and $G(P_v)$ yields $\tilde{S}$ and $\tilde{S}_v$ [$\bar{S}$ and $\bar{S}_v$],*
  - *S3. TLVM [ADVM] with seed matrix $\tilde{S}$ [$\bar{S}$] yields $\tilde{B}$ [$\bar{B}$], and*
  - *S4. Recovery using Algorithm 3.3 yields the solution of Equation (3.11) for unknown entries of ∇F, where we extract ∇F_c by knowing $P_c = P - P_v$.*
- *At all other iterations i.e. for $i > 1$ we perform the following last two steps of SJC2 as follows:*
  - *S3. TLVM [ADVM] with seed matrix $\tilde{S}_v$ [$\bar{S}_v$] yields $\tilde{D}$ [$\bar{D}$], and*
  - *S4. Recovery using Algorithm 3.4 yields the solution of Equation (3.12) for unknown entries of ∇F_v.*

The following introduces the conceptual recovery algorithms in steps S4 of all three variants of sparse accumulation processes SJC1, SJC2, and SJC3. Algorithm 3.3 for $mode = TLM$ recovers directly in forward mode the Jacobian ∇F from the compressed version $\tilde{B}$ for given sparsity pattern P and the seed matrix $\tilde{S}$. In similar manner, the same Jacobian can be recovered for $mode = ADJM$ from $\bar{B}$ in reverse mode for given P and $\bar{S}$. Let

$$color(i) = k \quad \text{for} \quad i \in \{1, \ldots, n\,[m]\} \quad \text{if} \quad \exists\, k \in \{1, \ldots, p\,[q]\} : \tilde{S}_{i,k} \neq 0\,[\,\bar{S}_{i,k} \neq 0\,]$$

denote the compressed column [row] index k of Jacobian column [row] i being the same as the color of the respective column [row] vertex in the bipartite graph $G(P)$ from which $\tilde{S}$ [$\bar{S}$] is obtained. Moreover,

$$group(k) = \{i \in \{1, \ldots, n\,[m]\} \mid color(i) = k\} \quad \text{for} \quad k \in \{1, \ldots, p\,[q]\}$$

denotes the set of all those columns [rows] of Jacobian that are compressed to the column [row] k of the compressed matrix $\tilde{B}$ [$\bar{B}$]. Likewise, one can recover ∇F directly using Algorithm 3.4 for $mode = TLM$ [$mode = ADJM$] and $p = p_v$ [$q = q_v$] in forward [reverse] mode from the compressed version $B = \tilde{D}$ [$B = \bar{D}$] for given $P = P_v$, $S = \tilde{S}_v$ [$\bar{S}_v$], and $\nabla F = \nabla F_c$.

Algorithm 3.3 (`JDR(mode, P, S, B, ∇F)` : Jacobian Direct Recovery)**.**

Require: Jacobian pattern P, seed matrix S, $mode = TLM$ resp. $mode = ADJM$ indicating column resp. row compression, compressed Jacobian B, and Jacobian $\nabla F = 0_{m \times n}$.
Ensure: the Jacobian matrix∇F with numerical values.

```
1: for j = 1 to m do
2:   for i = 1 to n do
3:     if P[j, i] ≠ 0 then
4:       if mode == TLM then
5:         ∇F[j, i] = B[j, color(i)]
6:       end if
7:       if mode == ADJM then
8:         ∇F[j, i] = B[color(j), i]
9:       end if
```

```
    end if
  end for
end for
```

Algorithm 3.4 (EJDR(mode, P, S, B, ∇F) : Enhanced Jacobian Direct Recovery)**.**

Require: Jacobian pattern P, seed matrix S, $mode = TLM$ resp. $mode = ADJM$ indicating column resp. row compression, compressed Jacobian B, and constant Jacobian $\nabla F = \nabla F_c$.
Ensure: the Jacobian matrix ∇F with numerical values.

```
for j = 1 to m do
  for i = 1 to n do
    if P[j, i] ≠ 0 then
      if mode == TLM then
        ∇F[j, i] = B[j, color(i)]
        for k ∈ group(color(i)) and k ≠ i do
          ∇F[j, i] −= ∇F[j, k]
        end for
      end if
      if mode == ADJM then
        ∇F[j, i] = B[color(j), i]
        for k ∈ group(color(j)) and k ≠ j do
          ∇F[j, i] −= ∇F[k, i]
        end for
      end if
    end if
  end for
end for
```

3.2.5 Case Study II : Sparse Hessian Computation

In the following and for the sake of simplicity, we assume that F of Equation (1.1) is scalar, that is, $n >> 1$ and $m = 1$. Hence, the Hessian

$$\mathbb{R}^{n \times n} \ni \nabla^2 F(\mathbf{x}) \equiv \left(\frac{\partial^2 y}{\partial x_j \partial x_i} \right)_{j,i=1,\ldots,n}$$

is a symmetric matrix of second-order partial derivatives that can be computed by application of SOTLM of F defined by Equation (1.10) at the computational cost of $O(n^2) \cdot Cost(F)$ by letting $\mathbf{x}^{(1)} = e_j$ and $\mathbf{x}^{(2)} = e_i$ range over Cartesian basis vectors $e_j, e_i \in D$ of the input domain $D \subseteq \mathbb{R}^n$ for $j, i = 1, \ldots, n$ such that

$$y^{(1,2)} =<< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >, \mathbf{x}^{(2)} >= \frac{\partial^2 y}{\partial x_j \partial x_i} \quad .$$

Likewise, the same Hessian can be computed using SOADM of F defined by Equation (1.11) at the computational cost of $O(n) \cdot Cost(F)$ by setting $y_{(1)} = 1$ and letting $\mathbf{x}^{(2)} = e_i$ range over Cartesian basis vectors $e_i \in D$ for $i = 1, \ldots, n$ such that

$$\mathbf{x}^{(2)}_{(1)} =< y_{(1)}, < \nabla^2 F, \mathbf{x}^{(2)} >>= \left(\frac{\partial^2 y}{\partial x_1 \partial x_i}, \ldots, \frac{\partial^2 y}{\partial x_n \partial x_i} \right)$$

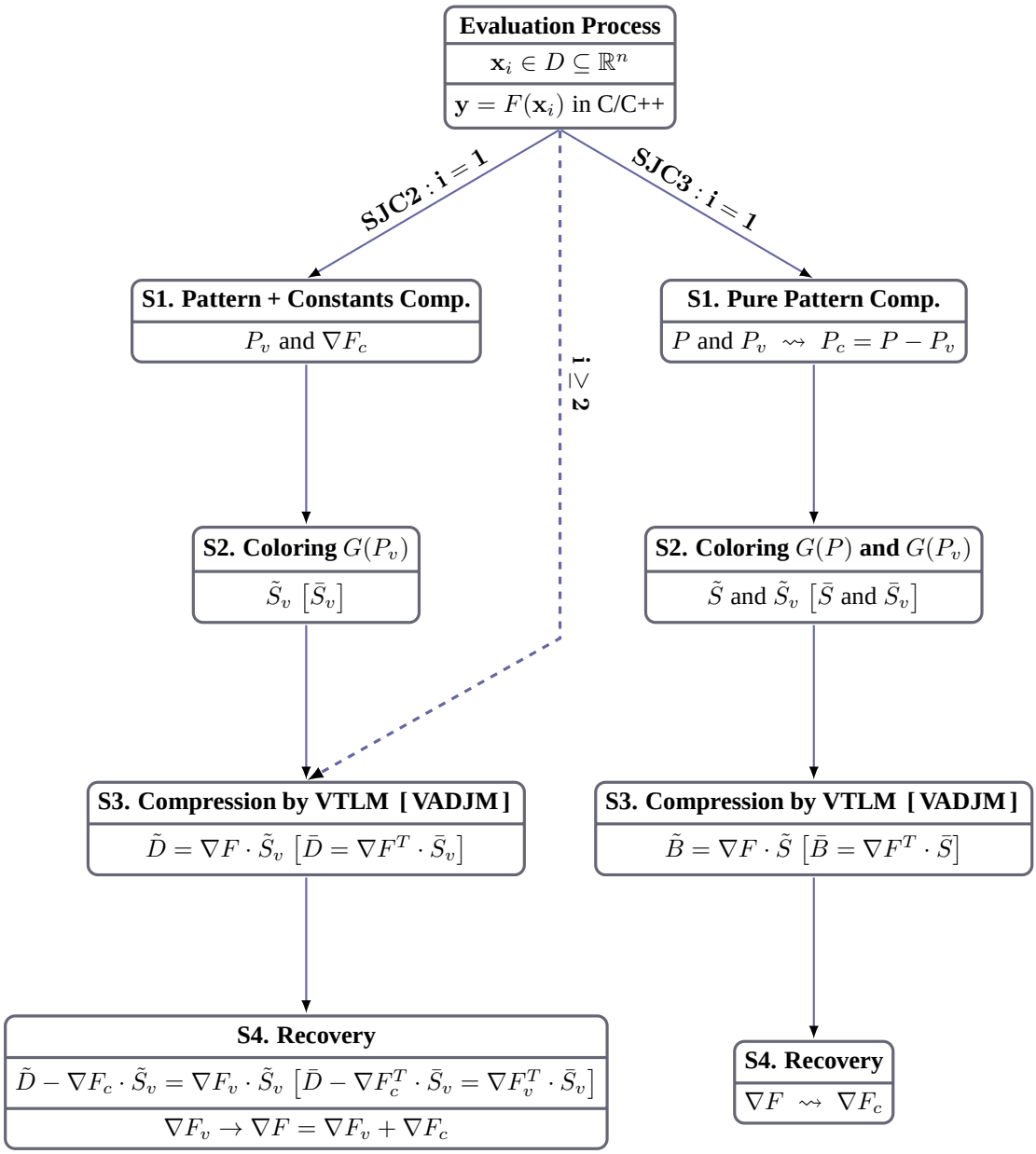


Figure 3.1: Entire Process of sparse Jacobian Computation with constant Exploitation.

representing the ith row of $\nabla^2 F$. We consider in the following SOADM as the model of choice for computing $\nabla^2 F(\mathbf{x})$ because of linear runtime complexity in n compared to quadratic one of SOTLM. Here, we consider SOADVM as a vector representation of SOADM that computes the compressed Hessian matrix

$$\mathbb{R}^{n\times p} \ni B = \nabla^2 F \cdot S \in \mathbb{R}^{n\times p} \tag{3.15}$$

by p times evaluation of SOADM of F in directions $\mathbf{x}^{(2)} = S_{*,i}$ for $i = 1, \ldots, p$ with S denoting the seed matrix resulting from the partitioning of columns of $\nabla^2 F$. The partitioning can be done in the same way as the Jacobian case by solving a graph coloring problem that is also known to be NP-complete [CM83, CC86] on the corresponding graph representation of $\nabla^2 F$. As shown by Coleman and Moré [CM83], hereby the seed matrix can be obtained by the application of either *star coloring* as a variant of distance-1 coloring with the restriction that every path over four vertices has to use at least three colors in combination with direct recovery or *acyclic coloring* as shown by Coleman and Cai [CC86] in combination with *indirect* (via substitution) recovery on the *adjacency graph* of $\nabla^2 F$. In the following we focus on the former and denote by $G(P^2)$ the adjacency graph of $\nabla^2 F$ obtained from its sparsity pattern

$$P^2 = P(\nabla^2 F) \equiv (p_{j,i})_{j,i=1,\ldots,n} \quad \text{with} \quad P_{j,i} \in \{0, 1\} \quad .$$

Thus, we consider SHC1 described in Procedure 3.4, which summarizes the classical process of sparse Hessian computation and assume, like in the Jacobian case, that we are interested in Hessians at points $\mathbf{x} \in I \subseteq D$ with fixed control flow of F in I. Algorithm 3.5, which is a modified version of `DIRECTRECOVER1` algorithm proposed by Gebremedhin et al. [GTPW09], is used to recover the Hessian entries from the compressed version.

Procedure 3.4. *The entire process of sparse Hessian computation (SHC1) is as follows:*

S1. Evaluation of F at the given point yields Hessian sparsity pattern P^2,

S2. Star coloring of $G(P^2)$ yields S,

S3. SOADVM with seed matrix S yields B, and

S4. Direct recovery using Algorithm 3.5 yields the solution of Equation (3.15) for unknown entries of $\nabla^2 F$.

Consequently,

$$color(i) = k \quad \text{for} \quad i \in \{1, \ldots, n\} \quad \text{if} \quad \exists\, k \in \{1, \ldots, p\} \quad \text{such that} \quad S_{i,k} \neq 0$$

denotes the compressed column index k of Hessian column i. Moreover,

$$group(k) = \{i \in \{1, \ldots, n\} : color(i) = k\} \quad \text{for} \quad k \in \{1, \ldots, p\}$$

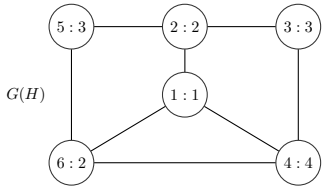
denotes the set of all those columns of Hessian that are compressed to column k of B.

Example 3.4. *In the following we consider the Hessian matrix H and its compressed version $B = H \cdot S$*

resulted from the application of SOADVM using the seed matrix S as follows.

$$\begin{pmatrix} h_{1,1} & h_{1,2}+h_{1,6} & 0 & h_{1,4} \\ h_{2,1} & h_{2,2} & h_{2,3}+h_{2,5} & 0 \\ 0 & h_{3,2} & h_{3,3} & h_{3,4} \\ h_{4,1} & h_{4,6} & h_{4,3} & h_{4,4} \\ 0 & h_{5,2}+h_{5,6} & h_{5,5} & 0 \\ h_{6,1} & h_{6,6} & h_{6,5} & h_{6,4} \end{pmatrix} = \begin{pmatrix} h_{1,1} & h_{1,2} & 0 & h_{1,4} & 0 & h_{1,6} \\ h_{2,1} & h_{2,2} & h_{2,3} & 0 & h_{2,5} & 0 \\ 0 & h_{3,2} & h_{3,3} & h_{3,4} & 0 & 0 \\ h_{4,1} & 0 & h_{4,3} & h_{4,4} & 0 & h_{4,6} \\ 0 & h_{5,2} & 0 & 0 & h_{5,5} & h_{5,6} \\ h_{6,1} & 0 & 0 & h_{6,4} & h_{6,5} & h_{6,6} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

S is the result of star coloring of the adjacency graph of H as follows.



The vertex labels of the shape $a : b$ indicate that vertex a has the color b. Hence, the coloring yields the following four groups of columns.

$$group(1) = \{1\}, \quad group(2) = \{2, 6\}, \quad group(3) = \{3, 5\}, \quad \textit{and} \quad group(4) = \{4\} \quad ,$$

where

$$color(1) = 1, \quad color(2) = color(6) = 2, \quad color(3) = color(5) = 3, \quad color(4) = 4 \quad .$$

Finally, the Hessian values are recovered by Algorithm 3.5. For instance, the entry $H[1, 2] = B[2, 1] = h_{2,1}$ is obtained in line 5 for $j = 1$ and $i = 2$ from B.

Constant Exploitation

In order to exploit the constants of the Hessian and to get better compression we consider the Hessian

$$\nabla^2 F = \nabla^2 F_c + \nabla^2 F_v$$

as the sum of its constants $\nabla^2 F_c$ and variable entries $\nabla^2 F_v$. We compute $\nabla^2 F_c$ along with the sparsity pattern $P_v^2 = P(\nabla^2 F_v)$ of $\nabla^2 F_v$ and obtain $S_v \in \mathbb{R}^{n \times q}$ by acyclic coloring of $G(P_v^2)$. Thus, application of SOADVM yields the compressed Hessian

$$D = \nabla^2 F \cdot S_v = \nabla^2 F_v \cdot S_v + \nabla^2 F_c \cdot S_v \quad . \tag{3.16}$$

Having D we obtain $\nabla^2 F_v$ by solving the linear system

$$D - \nabla^2 F_c \cdot S_v = \nabla^2 F_v \cdot S_v \tag{3.17}$$

using Algorithm 3.6 to recover column and row entries of $\nabla^2 F_v$. The entire process can be summarized as follows.

Procedure 3.5. *The entire process of sparse Hessian computation (SHC2) with constant exploitation:*

S1. Evaluation of F at the given point yields variable Hessian sparsity pattern P_v^2 and constant Hessian $\nabla^2 F_c$,

S2. Star coloring of $G(P_v^2)$ yields S_v,

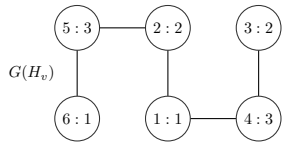
S3. SOADVM with seed matrix S_v yields D, and

S4. Direct recovery using Algorithm 3.6 yields the solution of Equation (3.15) for unknown entries of $\nabla^2 F_v$.

Example 3.5. *In the following we consider the Hessian matrix H of Example 3.5 with the constant entries $c_{1,6}, c_{6,1}, c_{2,3}, c_{3,2}, c_{4,6}, c_{6,4}$, and $c_{6,6}$. Thus, we get the compressed Hessian $D = H \cdot S_v$ by applying SOADVM using S_v as follows.*

$$\begin{pmatrix} h_{1,1}+c_{1,6} & h_{1,2} & h_{1,4} \\ h_{2,1} & h_{2,2}+c_{2,3} & h_{2,5} \\ 0 & c_{3,2}+h_{3,3} & h_{3,4} \\ h_{4,1}+c_{4,6} & h_{4,3} & h_{4,4} \\ h_{5,6} & h_{5,2} & h_{5,5} \\ c_{6,1}+c_{6,6} & 0 & h_{6,5}+c_{6,4} \end{pmatrix} = \begin{pmatrix} h_{1,1} & h_{1,2} & 0 & h_{1,4} & 0 & c_{1,6} \\ h_{2,1} & h_{2,2} & c_{2,3} & 0 & h_{2,5} & 0 \\ 0 & c_{3,2} & h_{3,3} & h_{3,4} & 0 & 0 \\ h_{4,1} & 0 & h_{4,3} & h_{4,4} & 0 & c_{4,6} \\ 0 & h_{5,2} & 0 & 0 & h_{5,5} & h_{5,6} \\ h_{6,1} & 0 & 0 & c_{6,4} & h_{6,5} & c_{6,6} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

S_v is the result of star coloring of the adjacency graph of H_v as follows.



Thereby, we get three as total number of used colors, which is better than four in case of coloring the graph of H. The three groups of columns are as

$$group(1) = \{1, 6\}, \quad group(2) = \{2, 3\}, \quad \textit{and} \quad group(3) = \{4, 5\} \ ,$$

where

$$color(1) = color(6) = 1, \quad color(2) = color(3) = 2, \quad color(4) = color(5) = 3 \quad .$$

Finally, the Hessian variable entries are recovered using Algorithm 3.6. For instance, $H[2, 2]$ is obtained for $j = 2$ and $i = 2$ from $D[2, 2]$ according to line 7. However, in order to get the right value of $H[2, 2]$ the constant value $\nabla^2 F_c[2, 3] = c_{2,3}$ for $k = 3$ according to line 10 is subtracted from $D[2, 2] = h_{2,2} + c_{2,3}$ yielding $H[2, 2] = D[2, 2] - c_{2,3} = h_{2,2}$. As one can see $D[6, 1] = c_{6,1} + c_{6,6}$ is a pure sum of constant entries, that obviously can be ignored in recovery step as the involved constants are already known. Thus, the recovery routine need only to care about recovering variable elements.

Like the Jacobian sparsity exploitation process SJC3 it may also pay off to obtain constants without the overhead of computing them on the SAC of F at runtime as illustrated in Figure 3.2. Therefore, first, at the starting point $\mathbf{x}_1$, we obtain the Hessian in the classical way as described in SHC1 of Procedure 3.4 with only difference that we also propagate P_v^2 along with P. Obviously, we get the constant pattern as $P_c^2 = P^2 - P_v^2$. After termination of the recovery step S4 yielding $\nabla^2 F(\mathbf{x}_1)$ and since we have P_c^2 we obtain easily $\nabla^2 F_c$ from $\nabla^2 F$. Furthermore, we obtain in step S2 both seed matrices S and S_v by star

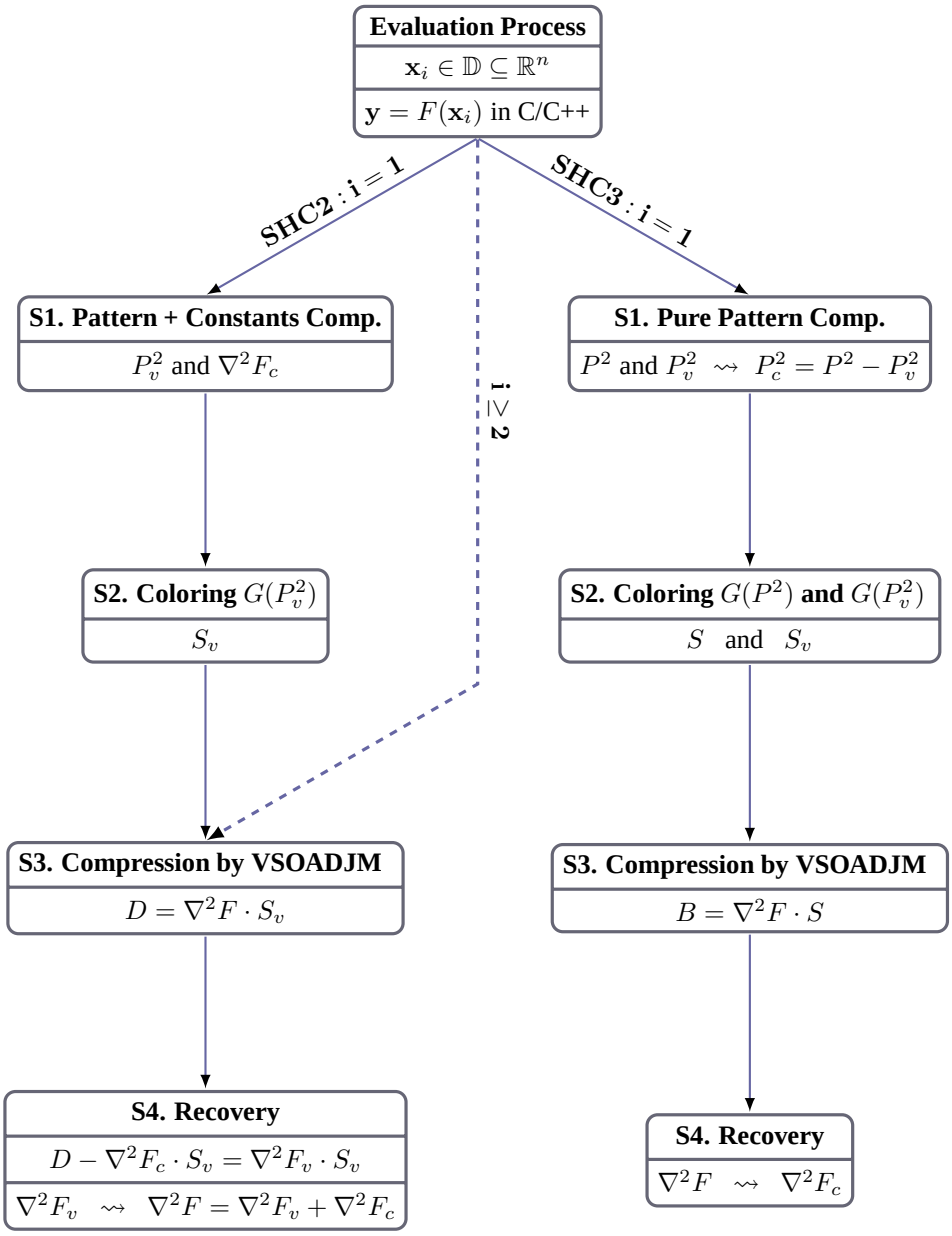


Figure 3.2: Entire Process of sparse Hessian Computation with constant Exploitation.

coloring of vertices of $G(P^2)$ and $G(P_v^2)$, respectively such that in the following iterations we proceed to accumulate the target Hessians just by performing the steps S3 and S4 of SHC2 shown in the left column of the same figure. Here, we assume that the termination of the leaf processes S4 of both left and right columns are followed by a jump to the root process, is the Hessian at another point is of interest. Otherwise, leaf processes are supposed to finalize the entire sparse accumulation process.

Procedure 3.6. *The alternative way to compute and exploit the Hessian constants (SHC3):*

- *At first iteration i.e. for $i = 1$ we perform an extended version of SHC1 as follows:*
  - *S1. Evaluation of F at the given point yields P^2 and P_v^2,*
  - *S2. Star coloring of $G(P^2)$ and $G(P_v^2)$ yields S and S_v,*
  - *S3. SOADVM with seed matrix S yields B, and*
  - *S4. Recovery using Algorithm 3.5 yields the solution of Equation (3.16) for unknown entries of $\nabla^2 F$, where we extract $\nabla^2 F_c$ by knowing $P_c^2 = P^2 - P_v^2$.*
- *At all other iterations i.e. for $i > 2$ we perform the following last two steps of SHC2 as follows:*
  - *S3. ADVM with seed matrix S_v yields D, and*
  - *S4. Recovery using Algorithm 3.6 yields the solution of Equation (3.17) for unknown entries of $\nabla^2 F_v$.*

Algorithm 3.5 ($\texttt{HDR}\,(\texttt{P}^2, \texttt{S}, \texttt{B}, \nabla^2\texttt{F})$: Hessian Direct Recovery)**.**

Require: : the Hessian pattern P^2, the seed matrix $S \in \mathbb{R}^{n\times p}$, the compressed Hessian $B \in R^{n\times p}$, and zero Hessian $\nabla^2 F = 0_{n\times n}$.
Ensure: : the Hessian matrix $\nabla^2 F$ with numerical values.

1: **for** $j = 1$ **to** n **do**
2: **for** $i = 1$ **to** n **do**
3: **if** $P[j, i] \neq 0$ **then**
4: **if** $\exists k \neq i$ **and** $P[j, k] \neq 0$ **and** $color(k) = color(i)$ **then**
5: $\nabla^2 F[j, i] = \nabla^2 F[i, j] = B[i, color(j)]$
6: **else**
7: $\nabla^2 F[j, i] = \nabla^2 F[i, j] = B[j, color(i)]$
8: **end if**
9: **end if**
10: **end for**
11: **end for**

Algorithm 3.6 ($\texttt{EHDR}\,(\texttt{P}_\texttt{v}^2, \texttt{P}_\texttt{c}^2, \texttt{S}_\texttt{v}, \texttt{D}, \nabla^2\texttt{F})$: Enhanced Hessian Direct Recovery)**.**

Require: : the variable resp. constant Hessian pattern P_v^2 resp. P_c^2, the seed matrix $S_v \in \mathbb{R}^{n\times q}$, the compressed Hessian $D \in \mathbb{R}^{n\times q}$, and the Hessian $\nabla^2 F := \nabla^2 F_c \in R^{n\times n}$ initialized to its constant part $\nabla^2 F_c$.
Ensure: : the Hessian matrix $\nabla^2 F$ with numerical values.

1: **for** $j = 1$ **to** n **do**

2: **for** $i = 1$ **to** n **do**
3: **if** $P_v[j,i] \neq 0$ **then**
4: **if** $\exists k \neq i$ **and** $P_v[j,k] \neq 0$ **and** $color(k) = color(i)$ **then**
5: $\nabla^2 F[j,i] = \nabla^2 F[i,j] = D[i, color(j)]$
6: **else**
7: $\nabla^2 F[j,i] = \nabla^2 F[i,j] = D[j, color(i)]$
8: **end if**
9: **for** $k \in group(color(i))$ **and** $P_c^2[j,k] \neq 0$ **do**
10: $\nabla^2 F[j,i] = \nabla^2 F[i,j] = \nabla^2 F[j,i] - \nabla^2 F[j,k]$
11: **end for**
12: **end if**
13: **end for**
14: **end for**

3.2.6 Numerical Results

In the following we present some numerical results on sparse computation of Jacobians of multivariate functions of type $F : \mathbb{R}^n \to \mathbb{R}^m$ such as Bratu of Listing 2.1 as well as the one that arises in Simulated Moving Bed (SMB) process a model for liquid chromatographic separation described by Gebremedhin et al. [GPW08], where in the former $n = m$. Moreover, we present results on sparse computation of the Hessian of the objective function of type $f : \mathbb{R}^n \to \mathbb{R}$ arising in SMB. Therefore, we use ColPack implementations of partial distance-2, acyclic, and star coloring algorithms with in ColPack terminology *natural* ordering of graph vertices. Moreover, we use TLVM and SOADM implementations provided by ADOL-C in order to accumulate accurate first and second order directional derivatives whenever appropriate.

The tests for Jacobian and Hessian compression with constant exploitation are performed using the C++ operator overloading tool *CompJacHess*[2]. The state-of-the-art implementation of the software computes simultaneously constants and variable pattern of Jacobian and Hessian of the underlying functions written in a subset of programming language C/C++. However, current implementation activities of the author focus on separating variable pattern estimation from constant retrieval that is needed in context of SJC3 and SHC3.

Constant Exploitation

Table 3.2 compares runtimes of four steps of sparse Jacobian and Hessian computation with (SJC2/SHC2) and without (SJC1/SHC1) exploitation of the constants. The resulting numbers of colors are given by column p. Here, p is the result of partial distance-2 and star coloring on the respective graphs of target Jacobians and Hessians, respectively. Let us consider SJC1 and SJC2 results of SMB presented in first two rows. As one can see the respective Jacobian with $n = 211755$ columns is compressed to a one with only eight in the former and six in the latter columns. Better compression is achieved on Bratu with a gain of six colors. Thus, in both cases the gain in runtime of step $S3$ is not considerably high, which we consider reasonable. We note that retrieving constants of Jacobian is of quadratic complexity as nonzero pattern for $d = 2$ are needed to specify variable entries as discussed on Algorithm 3.1. For the same reason the constant Hessian estimation is of cubic complexity. Les us consider now the sparse computation of Hessian of SMB objective function f for $n = 34305$ in the last two rows. Here, we observe that the star coloring underperforms in both runtime and achieved colors when we consider the variable Hessian (SHC2) compared to that of the nonzero one, despite the fact that the former is sparser as roughly 12 % of the Hessian nonzeros turn out to be constant. In this context, we consider the coloring

[2]CompJacHess stands for Compressed Jacobian and Hessian Computation

Mode	n	$T(S1)$	$T(S2)$	p	$T(S3)$	$T(S4)$	$\%cnz$
SJC1 (SMB)	211755	0.63	1.77	8	0.23	0.07	27.9
SJC2 (SMB)	211755	2.05	1.15	6	0.2	0.05	#
SHC1 (SMB)	34305	6.58	161.87	12364			12.1
SHC2 (SMB)	34305	1757.72	213.9	14401			
SJC1 (Bratu)	4000000	13.03	33.04	7	4.09	1.42	79.9
SJC2 (Bratu)	4000000	51.22	14.93	1	2.94	0.47	#

Table 3.2: Runtime and Coloring Results on SJC and SHC.

a major obstacle toward achieving better compression by constant exploitation of Hessians. In fact, we observe the same behavior when comparing the coloring results of EHP (i.e. nonzero pattern) with that of an conservative overestimated version of the same Hessian, whose coloring seems to perform much better compared to EHP without noticeable loss of runtime as described in the following.

3.3 Conservative Hessian Pattern Estimation

In the following we propose a method for *conservative overestimation* of the Hessian sparsity pattern (CHP). Furthermore, we compare its runtime behavior with that of the ADOL-C implementation of the standard algorithm for exact Hessian pattern estimation (EHP) proposed by Walther in [Wal08]. Therefore, we introduce first the standard algorithm and prove its complexity. Moreover, we exploit the partial separability [Gay96, Wal08, GT82] in CHP to reduce the runtime complexity of sparsity pattern estimation that is known to be quadratic in the dimension of inputs n in worst case. We also prove the complexity of CHP, which is a light modification of the standard one.

Finally, we present a recursive algorithm for Hessian pattern estimation (RHP) , which is obtained by the reapplication of the partial separability to every element operation on the SAC of F. Therefore, not only *nonlinear components* of the outputs are of interest but also those of all SAC variables. RHP is supposed to yield exactly the same pattern as CHP at recursion level one. Moreover, CHP converges to EHP for sufficiently large recursion level.

3.3.1 Exact Hessian Pattern Estimation

In the following we introduce an algorithm for estimating the exact sparsity pattern $P^2 = P(\nabla^2 F)$ of the Hessian matrix $\nabla^2 F$, which is a simplified version of TSP described by Algorithm 3.1 for $d = 2$. Therefore, for every SAC variable v_k with $k \in V$ we define index and index pair set

$$fod(v_k) = P^1(v_k) \subseteq X \quad \text{and} \quad sod(v_k) = P^2(v_k) \subseteq X \times X$$

as the first- and second-order dependency sets of v_k on independents $\mathbf{x}$, respectively. EHP described in Algorithm 3.7 computes the exact second-order dependencies $sod(\mathbf{y})$ of the outputs $\mathbf{y}$ of F from the first-order dependencies $FoD(\mathbf{x})$ of its inputs $\mathbf{x}$ defined as

$$sod(\mathbf{y}) = \bigcup_{j=1}^{m} sod(y_j) \quad \text{and} \quad FoD(\mathbf{x}) = \big(fod(x_i) := \{i\}\big)_{i=1,\ldots,n}$$

on the SAC of F, respectively. The first-order dependencies of independent SAC variables in lines 1–4 are initialized to those of independents $\mathbf{x}$, where the corresponding second-order dependencies remain empty. The main computation effort is perfromed in lines 5–16, where $\cup, \bigcup$, and $\prod, \times$ represent

union and cross product of the corresponding (pair) sets, respectively. In lines 17–19 the second-order dependencies $sod(v_j)$ of each dependent SAC variable v_j is added to $sod(\mathbf{y})$.

Algorithm 3.7 (EHP $(\mathtt{SAC}(\mathtt{F}), \mathtt{FoD}(\mathbf{x}), \mathtt{sod}(\mathbf{y}))$: Exact Hessian Pattern Estimation)**.**

Require: SAC of F and first-order dependencies $FoD(\mathbf{x})$ of the inputs $\mathbf{x}$.
Ensure: second-order dependencies $sod(\mathbf{y})$ of the outputs $\mathbf{y}$.

1: **for** $i := 1$ **to** n **do**
2: $fod(v_i) = fod(x_i)$
3: $sod(v_i) = \emptyset$
4: **end for**
5: **for** $k = n+1, \ldots q$ **do**
6: $fod(v_k) = \bigcup_{i \prec k} fod(v_i)$
7: **if** $\varphi_k \in \{+\}$ **then**
8: $sod(v_k) = \bigcup_{i \prec k} sod(v_i)$
9: **end if**
10: **if** $\varphi_k \in \{*\}$ **then**
11: $sod(v_k) = \prod_{i \prec k} fod(v_i) \ \cup \ \bigcup_{i \prec k} sod(v_i)$
12: **end if**
13: **if** $\varphi_k \in \Phi_N$ **then**
14: $sod(v_k) = fod(v_k) \times fod(v_k)$
15: **end if**
16: **end for**
17: **for** $j = n + p + 1$ **to** q **do**
18: $sod(\mathbf{y}) = sod(\mathbf{y}) \cup sod(v_j)$
19: **end for**

One can easily see that the most costly operation in EHP is the cross product of lines 11 and 14, which are performed for $\varphi_k \in \{*\}$ and $\varphi_k \in \Phi_N$, respectively. This fact makes this algorithm have a quadratic complexity as proven by Theorem 3.3 that is in fact a light modification of the one proposed by Walther [Wal08] for $m = 1$. Note that this complexity would also follows from TSp for $d = 1$.

Theorem 3.3. *Given SAC of F defined in Equation (*1.2*) and let $OPS(\mathtt{EHP})$ denote the operation count needed for Algorithm 3.7. Hence, we have*

$$OPS(EHP) \leq OPS(F) \cdot O(\hat{n}^2) \quad ,$$

where $OPS(F)$ is the number of floating point operations in SAC of F and

$$\hat{n} = \max_{j \in \{1,\ldots,m\}} \hat{n}_j \quad \textit{with} \quad \hat{n}_j = \max_{i \in X} \textit{ nonzero } (\nabla^2 F_{j,i,*})$$

denotes the maximum number of nonzeros per row over all rows of the Hessians $\nabla^2 F_j$.

Proof. Obviously, there exists a positive constant c such that

$$n \leq c \cdot \hat{n} \quad . \tag{3.18}$$

Furthermore, the number of elements in $fod(v_j)$ for $j \in Z \cup Y$ is bounded by $\hat{n}$ i.e.

$$|fod(v_j)| \in O(\hat{n}) \quad . \tag{3.19}$$

Thus, it follows that the union in line 6 can be performed in $O(\hat{n})$. From Equation (3.19) follows immediately that the cross products in lines 11 and 14 can be performed in $O(\hat{n}^2)$. At the same time we have

$$|sod(v_j)| \in O(\hat{n}^2)$$

representing the upper bound for the union operations performed in lines 8 and 11. □

We recapitulate here that the quadratic complexity of EHP is caused by the computation of $sod(v_k)$ as the cross product of $fod(v_i)$ and the union of $sod(v_i)$ of their arguments $i \prec k$ in case of nonlinear and linear operations, respectively. In the following section we introduce the conservative algorithm.

3.3.2 Exploitation of Partial Separability

In order to accelerate the Hessian pattern estimation we assume in the following that F is partially separable as

$$F(\mathbf{x}) = \sum_{i=1}^{|NF(F)|} f_i(\mathbf{x}) \quad \text{with} \quad NF(F) = \bigcup_{j \in Y} nf(v_j) \tag{3.20}$$

into nonlinear functions f_i, which we refer to as *nonlinear frontier* $NF = NF(F)$ components of F. Griewank and Toint [GT82] have shown that F is partially separable if $\nabla^2 F$ is sparse. Thus, differentiating F of Equation (3.20) with respect to $\mathbf{x}$ yields

$$\nabla^2 F(\mathbf{x}) = \sum_{i=1}^{|NF(F)|} \nabla^2 f_i(\mathbf{x}) \quad .$$

Thus, the exact and conservative sparsity pattern of $\nabla^2 F$ is given by

$$sod(\mathbf{y}) = \bigcup_{i=1}^{|NF(F)|} sod(f_i) \quad \text{and} \quad csod(\mathbf{y}) = \bigcup_{i=1}^{|NF(F)|} csod(f_i) \quad ,$$

respectively, where

$$csod(f_i) = fod(f_i)^2 = fod(f_i) \times fod(f_i)$$

denotes the *conservative second-order dependencies* of f_i on $\mathbf{x}$ with $sod(f_i) \subseteq csod(f_i)$. Thus, we can overestimate the sparsity pattern of $\nabla^2 F(\mathbf{x})$ first by computing $fod(f_i)$ of all NF components f_i followed by building a union of the self cross products $fod(f_i)^2$.

Algorithm 3.8 (CHP(SAC(F), FoD($\mathbf{x}$), csod($\mathbf{y}$)) : Conservative Hessian Pattern Estimation)**.**
Require: *SAC of F and first-order dependencies $FoD(\mathbf{x})$ of inputs $\mathbf{x}$.*
Ensure: *conservative second-order dependencies $sod(\mathbf{y})$ of the outputs $\mathbf{y}$.*

1: ***for*** $i = 1$ ***to*** n ***do***
2: $fod(v_i) = fod(x_i)$
3: $nf(v_i) = \emptyset$
4: ***end for***
5: ***for*** $k = n+1, \ldots q$ ***do***
6: $fod(v_k) = \bigcup_{i \prec k} fod(v_i)$
7: $nf(v_k) = \bigcup_{i \prec k} get\_nf(v_i)$
8: ***end for***
9: ***for*** $j = n+p+1$ ***to*** q ***do***

10: $csod(\mathbf{y}) = csod(\mathbf{y}) \cup \bigcup_{i \in nf(v_j)} fod(v_i)^2$
11: ***end for***

Algorithm 3.8 illustrates the computation of $csod(v_j)$ with $j \in Y$ of the outputs $\mathbf{y}$ on the SAC of F. Thereby, in addition to the computation of $fod(v_k)$ with $k \in V$, we propagate the NF set defined as

$$nf(v_k) = \bigcup_{i \prec k} nf(v_i) \quad \text{with} \quad get\_nf(v_i) = \begin{cases} nf(v_i) & \text{for } \varphi_i \in \{+\} \\ \{i\} & \text{for } \varphi_i \in \{*\} \cup \Phi_N \\ \emptyset & \text{otherwise} \end{cases} \quad . \tag{3.21}$$

Obviously, the NF of a SAC variable $k \in V - X$ results from the union of the NF of its arguments $i \prec k$ as shown in line 7. Therefore, a nonlinear argument is supposed to return itself as a NF component of v_k, whereas a linear one forwards its NF to v_k as defined by Equation (3.21). In other words, we aim to maintain a nonlinear frontier DAG (NF-DAG)

$$G_{NF} = (V_{NF}, E_{NF}) \tag{3.22}$$

with

$$V_{NF} = \{i \in V : \varphi_i \in \{*\} \cup \Phi_N\} \quad \text{and} \quad E_{NF} = \{(i,j) : i,j \in V_{NF} \text{ and } i \in nf(v_i)\} \tag{3.23}$$

at the time of evaluating F. Vertices represent the nonlinear elemental functions at the SAC of F, where each vertex $k \in V - X$ of NF-DAG represents v_k for $\varphi_k \in \Phi_N \cup \{*\}$. Therefore, vertex k is supposed to store $fod(v_i)$ of each of the arguments $i \prec k$ of v_k. Moreover, vertex k stores the information about its predecessor vertices for the NF-DAG denoted here by $nf(v_k)$. We emphasize here that the computationally expensive cross products along with their unions as shown in line 10 are performed only for output variables in the number of their NF components and not $OPS(F)$ as in EHP. Theorem 3.4 proves the computational complexity of CHP.

Theorem 3.4. *Given SAC of F of Equation (1.2) and let $OPS(CHP)$ denote the operation count needed for Algorithm 3.8. Hence, we have*

$$OPS(CHP) \leq |NF(F)| \cdot O(\hat{n}^2) + OPS(F) \cdot O(\hat{n} + N) \quad .$$

N denotes the maximum number of NF components overall elemental operations of F as

$$N = \max_{j \in V} |nf(v_j)| \quad . \tag{3.24}$$

Proof. As shown in Equation (3.18), there exists a positive constant c such that $n \leq c \cdot \hat{n}$. Furthermore, the numbers of elements in both $fod(v_j)$ and $nf(v_j)$ for $j \in V$ are bounded by $\hat{n}$ and N, respectively, such that

$$|fod(v_j)| \in O(\hat{n}) \quad \text{and} \quad |nf(v_j)| \in O(N) \quad . \tag{3.25}$$

Thus, it follows that the unions in lines 6 and 7 can be performed in $O(\hat{n} + N)$ operations. From Equation (3.25) it follows immediately that the cross products as well as the unions in line 10 can be performed in $O(\hat{n}^2)$ as $|csod(v_j)| \in O(\hat{n}^2)$. □

At this point it should be made clear that the overestimation is a side effect of treating a multiplication operation as an operation of type Φ_N such as $\sin$ and $\exp$. In fact, we can only obtain an overestimated

version of the Hessian pattern by CHP when multiplications are contained in the NF component of F. However, this is not alway the case as explained in the following example.

$\mathtt{v} = \exp(\mathtt{x}_1); \quad // \ \mathtt{fod}(\mathtt{v}) = \{1\}; \quad \mathtt{sod}(\mathtt{v}) = \{(1,1)\};$
$\mathtt{w} = \mathtt{v} * \mathtt{x}_1; \quad // \ \mathtt{fod}(\mathtt{w}) = \{1\}; \quad \mathtt{sod}(\mathtt{w}) = \mathtt{fod}(\mathtt{v}) \times \mathtt{fod}(\mathtt{x}_1) = \{(1,1)\};$
$\mathtt{y} = \mathtt{v} + \mathtt{w}; \quad // \ \mathtt{fod}(\mathtt{y}) = \{1\}; \quad \mathtt{sod}(\mathtt{y}) = \{(1,1)\}; \quad \mathtt{csod}(\mathtt{y}) = \mathtt{fod}(\mathtt{v})^2 \cup \mathtt{fod}(\mathtt{w})^2 = \{(1,1)\}$

The NF of y is obviously $nf(y) = \{v, w\}$. We obtain one and the same pattern by EHP and CHP as $fod(v) = fod(x_1)$, which implies that $fod(w)^2 = fod(v) \times fod(x_1)$. Hence, performing self cross product does not lead to any overestimation.

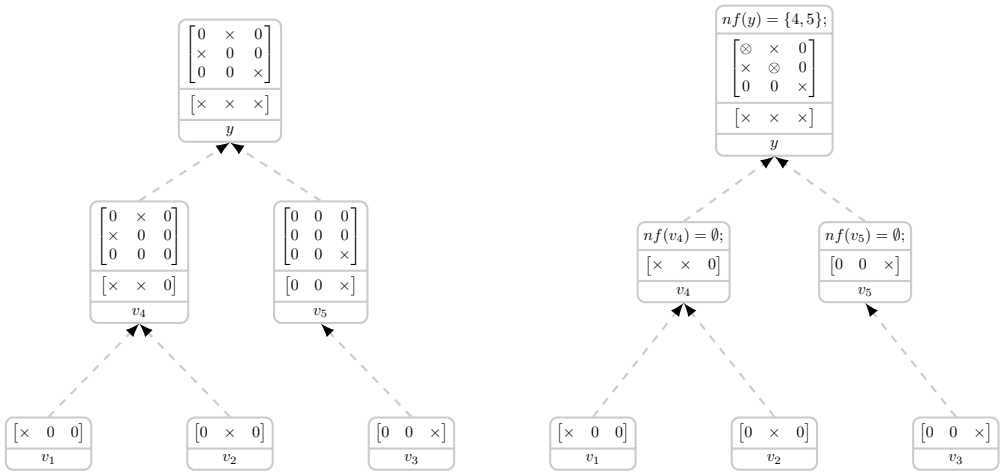
Example 3.6. *Given a scalar function $F : \mathbb{R}^2 \to \mathbb{R}$ defined as*

$$y = F(x_1, x_2, x_3) = x_1 \cdot x_2 + x_3^2 \quad . \tag{3.26}$$

We illustrate the algorithms EHP and CHP for exact and overestimate Hessian pattern computation on the following SAC of F. Here, we illustrate at every SAC statement $k = 1, \ldots, 5$, the computation of fod*s (a),* sod*s (b),* nf*s (c), and* csod*s (d).*

1. $v_i = x_i$ *for* $i = 1, 2, 3$
   - *(a)* $\mathtt{fod}(\mathtt{v}_\mathtt{i}) = \{\mathtt{i}\}$;
   - *(b)* $\mathtt{sod}(\mathtt{v}_\mathtt{i}) = \emptyset$;
   - *(c)* $\mathtt{nf}(\mathtt{v}_\mathtt{i}) = \emptyset$;
2. $v_4 = v_1 \cdot v_2$
   - *(a)* $\mathtt{fod}(\mathtt{v}_4) = \{1, 2\}$;
   - *(b)* $\mathtt{sod}(\mathtt{v}_4) = \mathtt{fod}(\mathtt{v}_1) \times \mathtt{fod}(\mathtt{v}_2) = \{(1, 2)\}$;
   - *(c)* $\mathtt{nf}(\mathtt{v}_4) = \emptyset$;
3. $v_5 = v_3^2$
   - *(a)* $\mathtt{fod}(\mathtt{v}_5) = \{3\}$;
   - *(b)* $\mathtt{sod}(\mathtt{v}_5) = \mathtt{fod}(\mathtt{v}_5) \times \mathtt{fod}(\mathtt{v}_5) = \{(3, 3)\}$;
   - *(c)* $\mathtt{nf}(\mathtt{v}_5) = \emptyset$;
4. $y = v_4 + v_5$
   - *(a)* $\mathtt{fod}(\mathtt{y}) = \{1, 2\}$;
   - *(b)* $\mathtt{sod}(\mathtt{y}) = \mathtt{sod}(\mathtt{v}_4) \cup \mathtt{sod}(\mathtt{v}_5) = \{(1, 2), (3, 3)\}$;
   - *(c)* $\mathtt{nf}(\mathtt{y}) = \mathtt{get\_nf}(\mathtt{v}_4) \cup \mathtt{get\_nf}(\mathtt{v}_5) = \{4, 5\}$;
   - *(d)* $\mathtt{csod}(\mathtt{y}) = \mathtt{fod}(\mathtt{v}_4) \times \mathtt{fod}(\mathtt{v}_4) \cup \mathtt{fod}(\mathtt{v}_5) \times \mathtt{fod}(\mathtt{v}_5) = \{(1, 1), (1, 2), (2, 2), (3, 3)\}$;

Considering the multiplication in (2). The second-order dependencies $\mathtt{sod}(\mathtt{v}_4)$ *of* v_4 *results from the cross product of* $\mathtt{fod}(\mathtt{v}_1)$ *and* $\mathtt{fod}(\mathtt{v}_2)$*. The nonlinear frontier of* v_4 *and* v_5 *are obviously empty as shown in (2) and (3), respectively. The nonlinear component of the addition operation* y *in (4) consists of its both nonlinear arguments, namely* 4 *and* 5*. Hence, the self cross product of the* $\mathtt{fod}(\mathtt{v}_4)$ *and* $\mathtt{fod}(\mathtt{v}_5)$ *of each nonlinear frontier component* 4 *and* 5 *of* y *followed by their union yields the overestimation of the second-order dependency* $\mathtt{csod}(\mathtt{y})$ *of the output* y *in (d) as* $(1, 1)$ *and* $(2, 2)$ *are not contained in* $\mathtt{sod}(\mathtt{y})$*. The entire process is shown graphically in Figure 3.3, where* fod *and* sod *along with* $csod$ *are denoted by dependency vectors and matrices, respectively. The exact and overestimated nonzeros are denoted by symbols* $\times$ *and* $\otimes$*, respectively.*



(a) (b)

Figure 3.3: Exact (a) and conservative (b) Hessian Pattern Estimation.

3.3.3 Recursive Hessian Pattern Estimation

As explained above the nonlinear frontier DAG G_{NF} of Equation (3.22) is supposed to be the result of evaluating F at given point $\mathbf{x}$. Each vertex is supposed to maintain its own nonlinear frontier given as its predecessors along with the fods of the arguments of the respective SAC statement.

Note that in CHP the focus was only the nonlinear frontier of outputs were of interest. There, performing self-cross products of the fods of each of the NF components resulted in the conservative overestimation of the Hessian pattern. However, given G_{NF} we can also formulate a recursive version of CHP that we refer to as recursive Hessian pattern estimation (RHP). Thus, one can formulate a recursive top-down algorithm on G_{NF} that is supposed to interpret nonlinear frontiers level-wise. The interpretation is nothing else than building cross products of fods of concerned nonlinear frontiers accordingly. Thus, we believe that with increasing levels (going further down on G_{NF}) RHP converges to EHP. A proof of concept implementation of this idea is attached to this work.

To clarify the idea behind RHP, let us consider again Figure 3.3 (b) that we obtain by interpreting the fods of the nonlinear components of the root box y. Now, let us go one level down to v_4 and v_5 and let assume that v_4 has access to the fods of its arguments v_1 and v_2. Now, despite the fact that v_4 is the result of the multiplication we can build the cross products of the fods of v_1 and v_2, yielding the pattern of the local Hessian as $P(\nabla^2 v_4) = fod(v_1) \times fod(v_2) = \{(1, 2)\}$. In case of v_5 nothing changes as its is the result of $\exp(v_3)$ and hence a self cross product of its fods yields the exact pattern as $P(\nabla^2 v_5) = \{(3, 3)\}$. Finally, the union $P(\nabla^2 y) = sod(v_4) \cup sod(v_5)$ yields the exact Hessian pattern of y.

We obtain RHP by replacing the statement in line 10 of CHP by $\texttt{compute}(\texttt{j}, \texttt{sod}(\mathbf{y}), \texttt{l})$, where $l \geq 0$ denotes the recursion level as described by Algorithm 3.9. j is supposed to denote the vertex corresponding to SAC variable v_j. We note that SAC variables are also supposed to have access to their NF components on the NF-DAG. More precisely, $nf(v_k)$ of the SAC variable v_k is supposed to point to the respective vertices on NF-DAG.

Thus, calling $\texttt{compute}(\texttt{j}, \texttt{P}, \texttt{l})$ adds the contribution of the vertex j to the second-order dependencies $P = sod(\mathbf{y})$ depending on the level l. The interpretation is performed in lines 2 and 9. In the former, the interpretation is performed in cases when $l = 0$ or $\varphi_j \in \Phi_N$. In this context, the interpretation is nothing else than building self cross product of the fod of the particular vertex. Otherwise, we proceed

recursively by calling $\texttt{compute}(\texttt{i}, \texttt{P}, \texttt{l}-1)$ for all the predecessors i of k as shown in lines 4–6. Finally, and after handling all the children of k, we also add the contribution of the multiplication (line 9) to P by building cross product of the fods of the parameter of v_k. We emphasize here that we explicitly distinguish between the parameters at SAC level and children on NF-DAG. The former do not necessarily represent nonlinear operations as opposed to the latter. This is exactly the reason for storing those fods during the evaluation of F in NF-DAG vertices. One can easily figure out that the $\texttt{compute}(\cdot)$ algorithm and hence the interpretation follows a **depth-first post-order** strategy.

Thus, we believe that with increase in levels (going further down on NF-DAG) RHP converges to EHP. This behavior is demonstrated using the proof of concept implementation of RHP in Section 3.3.4 on an artificial scalar function. The complexity class of computing the exact Hessian pattern by RHP is stated in Theorem 3.5. Nonetheless, further investigations are needed to handle the memory bandwidth as NF-DAG can potentially get very big. Hence, ideas are desired to reduce the memory consumption by freeing the memory of all unnecessary vertices of NF-DAG for a given recursion level to avoid running out of memory. Therefore, we suppose to keep track of the memory usage and delete as many vertices as possible whenever the memory bound is hit. As a first idea, it may make sense to mark a dead vertex i as not eliminatable if it is in a distance (number edges) $0 < dist(i,k) \leq l$ to at least one local dependent vertex[3] k. The marking is supposed to be performed by a breath-first traversing G_{NF}.

Theorem 3.5. *Given the nonlinear DAG $G_{NF} = (V_{NF}, E_{NF})$ defined by Equation (*3.22*) and let $OPS(RHP)$ denote the operation count needed for RHP described by Algorithm 3.9. Hence, we have*

$$OPS(RHP) \leq |V_{NF}| \cdot O(\hat{n}^2) + OPS(F) \cdot O(\hat{n} + N) \tag{3.27}$$

*with V_{NF} and N being defined by Equation (*3.23*) and Equation (*3.24*), respectively.*

Proof. As shown in Theorem 3.4 for CHP, the unions of lines 6–7 can be performed in $O(\hat{n} + N)$ operations for every SAC variable. From Equation (3.25) follows also that the cross products as well as the unions in lines 2 and 9 of the `compute` routine described by Algorithm 3.10 can be performed in $O(\hat{n}^2)$. We obtain Equation (3.27) taking into account that these operations are performed at most $|V_{NF}|$ times. □

Algorithm 3.9 ($\texttt{RHP}(\texttt{SAC}(\texttt{F}), \texttt{fod}(\mathbf{x}), \texttt{sod}(\mathbf{y}), \texttt{l})$: Recursive Hessian Pattern Estimation)**.**
Require: *SAC of F and recursion level $l \geq 0$.*
Ensure: *Second-order dependencies $sod(\mathbf{y})$ of the outputs $\mathbf{y}$.*

1: ***for*** $i := 1$ ***to*** n ***do***
2: $fod(v_i) = \{i\}$
3: $nf(v_i) = \emptyset$
4: ***end for***
5: ***for*** $k = n+1, \ldots q$ ***do***
6: $fod(v_k) = \bigcup_{i \prec k} fod(v_i)$
7: $nf(v_k) = \bigcup_{i \prec k} get\_nf(v_i)$
8: ***end for***
9: ***for*** $k = n+p+1$ ***to*** q ***do***
10: $\texttt{compute}(\texttt{v}_\texttt{k}, \texttt{sod}(\mathbf{y}), \texttt{l})$
11: ***end for***

Algorithm 3.10 ($\texttt{compute}(\texttt{k}, \texttt{P}, \texttt{l})$: Computation of Second-Order Dependencies)**.**

[3] A vertex in NF-DAG is marked as local dependent, if it has no successors.

Require: *NF-DAG vertex k with $k \in V$ of F and recursion level $l \geq 0$.*
Ensure: *contribution of v_k at level l to $P \subseteq X \times X$.*

1: ***if*** $l == 0$ ***or*** $\varphi_k \in \Phi_N$ ***then***
2: $P = P \cup (\bigcup_{i \prec k} fod(v_i))^2$
3: ***else***
4: ***for all*** $i \in nf(v_k)$ ***do***
5: $\texttt{compute}(\texttt{i}, \texttt{P}, \texttt{l} - \texttt{1})$
6: ***end for***
7: ***end if***
8: ***if*** $\varphi_k \in \{*\}$ ***then***
9: $P = P \cup \prod_{i \prec k} fod(v_i)$
10: ***end if***

In the following we illustrate RHP in the example function of Equation (3.26). The respective exact and conservative overestimated pattern by EHP and CHP, respectively, have already been illustrated in Example 3.6.

Example 3.7. *Given the scalar function $F : \mathbb{R}^2 \to \mathbb{R}$ defined by Equation (3.26) we illustrate RHP on the following SAC of F. Here, we illustrate at every SAC statement $k = 1, \ldots, 5$ the computation of* `fod`*s (a) as well as* `nf`*s (b). Moreover, the computation of $sod(y)$ is exercised for $l = 1$ and $l = 2$ in (c) and (d), respectively.*

1. $v_i = x_i$ *for* $i = 1, 2, 3$
   - *(a)* $\texttt{fod}(\texttt{v}_\texttt{i}) = \{\texttt{i}\}$;
   - *(b)* $\texttt{nf}(\texttt{v}_\texttt{i}) = \emptyset$;
2. $v_4 = v_1 \cdot v_2$
   - *(a)* $\texttt{fod}(\texttt{v}_\texttt{4}) = \{\texttt{1}, \texttt{2}\}$;
   - *(b)* $\texttt{nf}(\texttt{v}_\texttt{4}) = \emptyset$;
3. $v_5 = v_3^2$
   - *(a)* $\texttt{fod}(\texttt{v}_\texttt{5}) = \{\texttt{3}\}$;
   - *(b)* $\texttt{nf}(\texttt{v}_\texttt{5}) = \emptyset$;
4. $y = v_4 + v_5$
   - *(a)* $\texttt{fod}(\texttt{y}) = \{\texttt{1}, \texttt{2}\}$;
   - *(b)* $\texttt{nf}(\texttt{y}) = \texttt{get\_nf}(\texttt{v}_\texttt{4}) \cup \texttt{get\_nf}(\texttt{v}_\texttt{5}) = \{\texttt{4}, \texttt{5}\}$;
   - *(c)* $\texttt{compute}(\texttt{y}, \texttt{sod}(\texttt{y}), \texttt{2})$ *with initial* $sod(y) = \emptyset$ *and* $l = 2$

$$
\begin{aligned}
& compute(4, sod(y), 1) \\
\rightarrow \quad & \texttt{sod}(\texttt{y}) = \texttt{sod}(\texttt{y}) \cup \texttt{fod}(\texttt{v}_\texttt{1}) \times \texttt{fod}(\texttt{v}_\texttt{2}) = \{(\texttt{1}, \texttt{2})\} \\
& compute(5, sod(y), 1) \\
\rightarrow \quad & \texttt{sod}(\texttt{y}) = \texttt{sod}(\texttt{y}) \cup \texttt{fod}(\texttt{v}_\texttt{5}) \times \texttt{fod}(\texttt{v}_\texttt{5}) = \{(\texttt{1}, \texttt{2}), (\texttt{3}, \texttt{3})\}
\end{aligned}
$$

(d) $\texttt{compute}(\texttt{y}, \texttt{sod}(\texttt{y}), 1)$ *with initial* $sod(y) = \emptyset$ *and* $l = 1$

$$\begin{aligned}
&\texttt{compute}(4, \texttt{sod}(\texttt{y}), 0)\\
&\qquad \rightarrow \quad \texttt{sod}(\texttt{y}) = \texttt{sod}(\texttt{y}) \cup \texttt{fod}(\texttt{v}_4) \times \texttt{fod}(\texttt{v}_4) = \{(\mathbf{1}, \mathbf{1}), (1, 2), (\mathbf{2}, \mathbf{2})\}\\
&\texttt{compute}(5, \texttt{sod}(\texttt{y}), 0)\\
&\qquad \rightarrow \quad \texttt{sod}(\texttt{y}) = \texttt{sod}(\texttt{y}) \cup \texttt{fod}(\texttt{v}_5) \times \texttt{fod}(\texttt{v}_5) = \{(\mathbf{1}, \mathbf{1}), (1, 2), (\mathbf{2}, \mathbf{2}), (3, 3)\}
\end{aligned}$$

Hence, RHP results in overestimated and exact Hessian patterns for $l = 1$ *and* $l = 2$*, respectively. The overestimated pairs are marked as bold.*

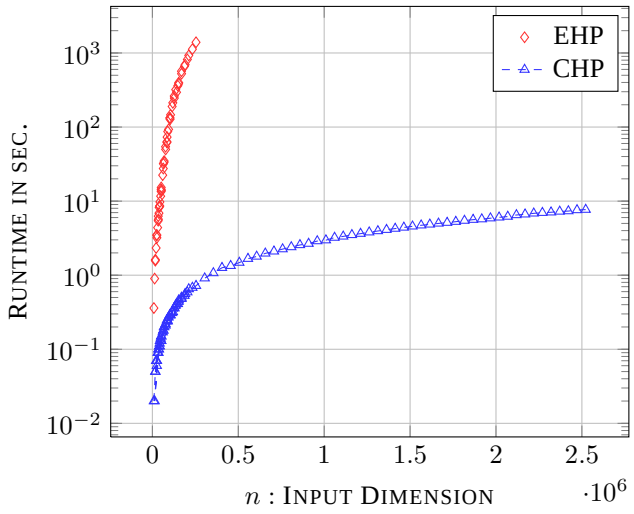
3.3.4 Numerical Results

Hessian Pattern Overestimation

In the following we consider numerical result on Hessian pattern overestimation. Figure 3.4 (a) compares the runtime of the conservative algorithm CHP against that of EHP implemented in AD tool ADOL-C on Hessian matrices of the objective function f of SMB. We observe here a linear growth in input dimension n in case of CHP, whereas EHP tends to increase quadratically with n. We observe also that the resulting colors q_s via star coloring of the conservative sparsity pattern resulting from CHP as shown in Figure 3.4 (b) is much better than the one obtained from EHP shown in (c) with a loss in runtime by a factor smaller than two in the former.

For instance, let us consider $n = 16980$. Star coloring of EHP takes 37.86 seconds to yield 6094 colors, whereas it takes 43.74 seconds to yield 37 colors in case of CHP as shown in columns T_s and q_s of (c) and (b), respectively. Moreover, we observe roughly the same behavior in case of the acyclic coloring algorithm with even better runtime in CHP as shown in columns T_a and q_a. Considering again $n = 16980$, acyclic coloring of CHP yields 5 colors in 58.63 seconds instead of 4861 colors in 88.14 in EHP. Thus, conservative estimation of the Hessian pattern seems to reduce the number of colors and hence improve the compression resulting from both star and acyclic coloring heuristics drastically without significantly affecting the runtime. For the sake of completeness we provide in columns T_p and q_p the runtime and coloring results of partial distance-2 coloring algorithm without symmetry exploitation in both EHP and CHP. Considering $n = 16980$ again distance-2 coloring of EHP yields 7277 colors in 4.76 seconds. Hence, the gain in time compared with star [acyclic] coloring yielding 6094 [4861] colors is roughly $8 \approx \frac{37.86}{4.76}$ [$9 \approx \frac{43.74}{4.76}$] by a loss of $1.2 \approx \frac{7277}{6094}$ [$1.5 \approx \frac{7277}{4861}$] in the number of achieved colors.

In conclusion we note here that both heuristics do sequential coloring, which we conjecture to be the reason for different color results of EHP and CHP. Therefore, we consider in the following a snapshot of the Hessian pattern of f for $n = 411$ as shown in Figure 3.5. The labels $i : j$ denote that vertex i gets the color j. The Symbol $\otimes$ denotes an overestimated entry of the Hessian. Firstly, let us focus on first ten rows and columns of EHP and the resulting adjacency graph $G(EHP)$. Star coloring results in a total of four colors, where the vertices 1-5 and 6 get the color 1 and 2, respectively. The latter is the case because vertex 6 is directly connected to 1. Moreover, vertex 7 has to be colored as 3 since otherwise the four vertices 6, 1, 7, and 4 connected by a path would have two colors and not at least three, which is required by star coloring. In fact the same argumentation holds for vertex 4. Consider now the entire pattern including the row/column 11 and its adjacency graph with vertex 11 and its incident edges. This vertex exhibits exactly the same property as vertices 3 and 4, thereby increasing the color number by one. Furthermore, we observe the same behavior by changing the ordering of vertices with respect to their degrees equal the number of incident vertices. This is because most vertices here are almost of the same degree. In opposite, star coloring of CHP first colors vertices $i = 1, 2, 3, 4$ with i while coloring most of the remaining vertices with one of the used colors.



(a)

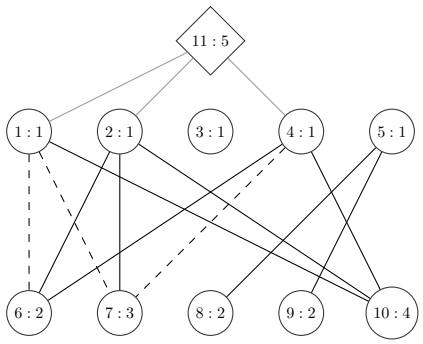
n	T_p	q_p	T_s	q_s	T_a	q_a
8580	1.25	3677	10.49	37	14.06	5
10680	1.93	4577	16.447	7	21.96	5
12780	2.78	5477	23.64	7	31.63	5
14880	3.73	6377	33.4	37	44.49	5
16980	**4.85**	**7277**	**43.74**	**37**	**58.63**	**5**
19080	6.13	8177	55.25	7	73.99	5
21180	7.55	9077	68.7	7	91.54	5
25555	10.91	10952	100.11	7	133.83	5
29755	14.85	12752	140.09	67	186.44	5
33955	19.33	14552	182.94	67	242.86	5
38155	24.28	16352	231.26	7	307.89	5

(b)

n	T_p	q_p	T_s	q_s	T_a	q_a
8580	1.22	3677	9.13	3094	19.4	2461
10680	1.9	4577	14.31	3844	30.47	3061
12780	2.69	5477	21.04	4594	44.78	3661
14880	3.65	6377	28.7	5344	61.39	4261
16980	**4.76**	**7277**	**37.86**	**6094**	**80.14**	**4861**
19080	5.98	8177	47.68	6844	101.68	5461
21180	7.35	9077	59.51	7594	126.55	6061
25555	10.68	10952	88.69	9184	188.94	7321
29755	14.45	12752	120.1	10684	257.86	8521
33955	18.83	14552	158.45	12184	338.73	9721
38155	23.77	16352	200.5	13684	432.43	10921

(c)

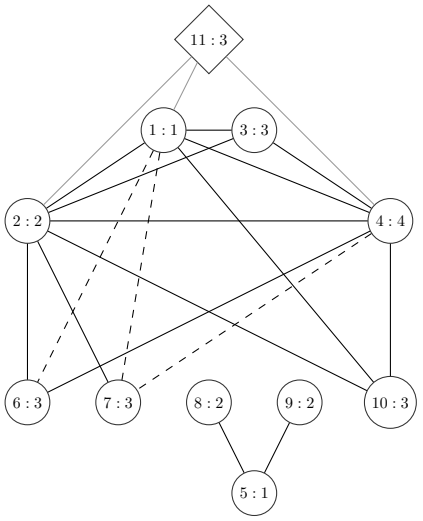
Figure 3.4: Runtime Comparison (a) and Coloring Results on CHP (b) and EHP (c) on SMB.



$G(EHP)$

	1	2	3	4	5	6	7	8	9	10	11
1						×	×			×	×
2						×	×			×	×
3											
4						×	×			×	×
5					×			×	×		
6	×	×		×		×					
7	×	×		×			×				
8					×						
9					×						
10	×	×		×						×	
11	×	×		×							×

$G(EHP)$



$G(CHP)$

	1	2	3	4	5	6	7	8	9	10	11
1	⊗	⊗	⊗	⊗		×	×			×	×
2	⊗	⊗	⊗	⊗		×	×			×	×
3	⊗	⊗	⊗	⊗							
4	⊗	⊗	⊗	⊗		×	×			×	×
5					×			×	×		
6	×	×		×		×					
7	×	×		×			×				
8					×			⊗			
9					×				⊗		
10	×	×		×						×	
11	×	×		×							×

$G(CHP)$

Figure 3.5: Star Coloring of the Adjacency Graph of CHP and EHP of SMB.

Recursive Hessian Pattern Estimation

As last test case we consider the scalar function $f : \mathbb{R}^n \to \mathbb{R}$ implemented in Listing 3.3 mapping $n = \text{pow}(2, h)$ independents stored in x[0][] onto y while performing non-overlapping pairwise multiplication [addition] of two consecutive entries of x[i][] for uneven [even] values of $i \in 1, \ldots, h$.

Note that the number of additions performed is almost half the number of multiplications. This becomes important when comparing the runtime of estimating the exact Hessian pattern using EHP implemented by ADOL-C and the one gained by RHP. In particular, the observed gain of factor two by the latter is due to the fact that the NF-DAG as defined by Equation (3.22) consists of vertices representing nonlinear operations. Hence, the union of second-order dependency sets are avoided in RHP for linear operations as opposed to EHP as described by Algorithm 3.7.

For illustration, let us assume for the time being $h = 2$. Hence, for $i = 0$ line 32 of Listing 3.3 results in

$$\texttt{x[1][0]=x[0][0]*x[0][1]} \quad \text{and} \quad \texttt{x[1][1]=x[0][2]*x[0][3]} \quad .$$

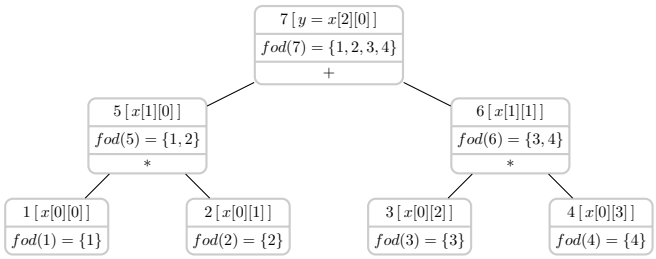
Additionally, for $i = 1$ line 33 yields x[2][0]=x[1][0]+x[1][1] denoting the output y. Hence, two multiplications and one addition are performed in total as returned by f in line 36. The former and latter are counted in lines 7 and 17, respectively. Obviously, $n = 4$ as $n = \text{pow}(2, h) = 2^2$.

Listing 3.3: Artificial Scalar Function f

```
1 // multiplicates pairwise entries of s and stores the results in t
2 // returns the number of performed multiplications
3 int multiply(int n, double* s, double* t) {
4   int muls=0;
5   for (int j=0; j<(n/2); j++) {
6     t[j]=s[2*j]*s[2*j+1];
7     muls++;
8   }
9   return muls;
10 }
11 // adds pairwise entries of s and stores the results in t
12 // returns the number of performed additions
13 int add(int n, double* s, double* t) {
14   int adds=0;
15   for (int j=0; j<(n/2); j++) {
16     t[j]=s[2*j]+s[2*j+1];
17     adds++;
18   }
19   return adds;
20 }
21 // x[0][] and y denote independents and dependents

22 // n and h denote the number of independents and Computational Graph Height

23 // returns the total number of performed multiplications and additions

24 int f(int h, double**& x, double& y) {
25   int muls=0, adds=0;
26   int n = pow(2,h);
27   x = new double* [h+1];
28   for (int i=0; i<=h; i++)
29     x[i] = new double [(int) pow(2, h-i)];
30
```

```
31 for (int i=0; i<h; i++) {
32     if (i%2==0)  muls+=multiply(x[i], x[i+1], pow(2, h-i));
33     else  adds+=add(x[i], x[i+1], pow(2, h-i));
34   }
35   y = x[h][0];
36   return (muls+adds);
37 }
```

The computational graph $G(f)$ of f resulting for $h = 2$ is shown in Figure 3.6 (a). It is a balanced tree of height h with leaves [root] denoting the independents [dependent] vertices [vertex]. Note that NF-DAG of f would only consist of two vertices 5 and 6 being NF components of the output y. In this case, vertices 5 and 6 have to store the first-order dependencies of the arguments of the respective SAC statements. In particular, vertex 5 [6] has access to $fod(1)$ and $fod(2)$ [$fod(3)$ and $fod(4)$].



(a) : $G(f)$ for $h = 2$

	1	2	3	4
1	0	×	0	0
2	×	0	0	0
3	0	0	0	×
4	0	0	×	0

(b) : EHP

	1	2	3	4
1	⊗	×	⊗	⊗
2	×	⊗	⊗	⊗
3	⊗	⊗	⊗	×
4	⊗	⊗	×	⊗

(c) : RHP(0)

	1	2	3	4
1	⊗	×	0	0
2	×	⊗	0	0
3	0	0	⊗	×
4	0	0	×	⊗

(d) : RHP(1)

	1	2	3	4
1	0	×	0	0
2	×	0	0	0
3	0	0	0	×
4	0	0	×	0

(e) : RHP(2)

Figure 3.6: Computational Graph of f for $h = 2$ (a) and the resulting sparsity Pattern from EHP (b) and RHP for Recursions 0 (c), 1 (d) and 2 (e). The symbols $\times$ and $\otimes$ denote exact and overestimated pattern entries, respectively.

Figure 3.6 (b) shows the resulting exact sparsity pattern of the Hessian of f for $h = 2$ obtained by EHP. The resulting sparsity pattern from RHP for recursion levels $0, 1$, and 2 are given by (c), (d) and (e), respectively. For a given recursion level l, the resulting sparsity obtained by RHP is denoted by RHP(l). Obviously, RHP(2) in (e) looks the same as (b), which illustrates the convergence of RHP to EHP for recursion level 2. One can figure out that the convergence is achieved for $l = \frac{h}{2} + 1$ for this example function. Moreover, RHP(1) in (d) would also result from CHP as two multiplications of vertices 5 and 6 in (a) are the NF component of y as already mentioned.

More precisely, (c) results from RHP for $l = 0$ according to line 2 of Algorithm 3.10 by self cross product

$$fod(7) \times fod(7) = \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \quad .$$

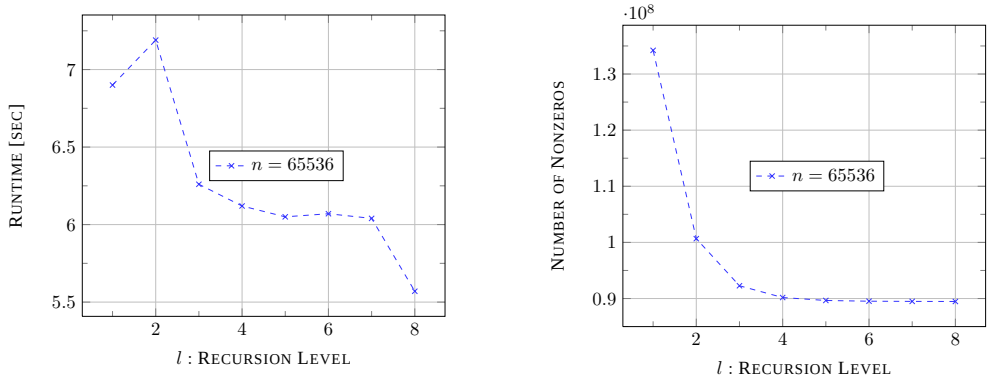
However, choosing $l = 1$ would lead to the call of the `compute`$(\cdot)$ with recursion $l = 0$ on 5 and 6 as NF components of y. Hence, self cross products

$$fod(5) \times fod(5) = \{1, 2\} \times \{1, 2\} \quad \text{and} \quad fod(6) \times fod(6) = \{3, 4\} \times \{3, 4\}$$

and their unions yields (d). Finally, in case of $l = 2$ the exact Hessian pattern is estimated as the recursion level reached at vertices 5 and 6 is 1. Hence and as none of them has any NF component the cross products

$$fod(1) \times fod(2) = \{1\} \times \{2\} \quad \text{and} \quad fod(3) \times fod(4) = \{3\} \times \{4\}$$

followed by their unions are performed according to the line 9 of Algorithm 3.10. Thus, the overestimated diagonal entries of (d) are removed yielding (e). Obviously, starting from (c), the exact Hessian pattern (2) is successively approximated by increasing recursion level. Figure 3.7 presents first runtime results on



(a) (b)

$n = 4096$	nz	$T(P)$	T_a	q_a
RHP(1)	8388608	0.38	501.48	2048
RHP(2)	6291456	0.41	458.3	1536
RHP(3)	5767168	0.36	289.91	1408
RHP(4)	5636096	0.35	237.58	1376
RHP(5)	5603328	0.34	223.91	1368
RHP(6)	**5595136**	**0.33**	**220.16**	**1366**
RHP(7)	5591040	0.3	220.27	1366
EHP	**5591040**	**0.82**	**220.65**	**1366**

(c)

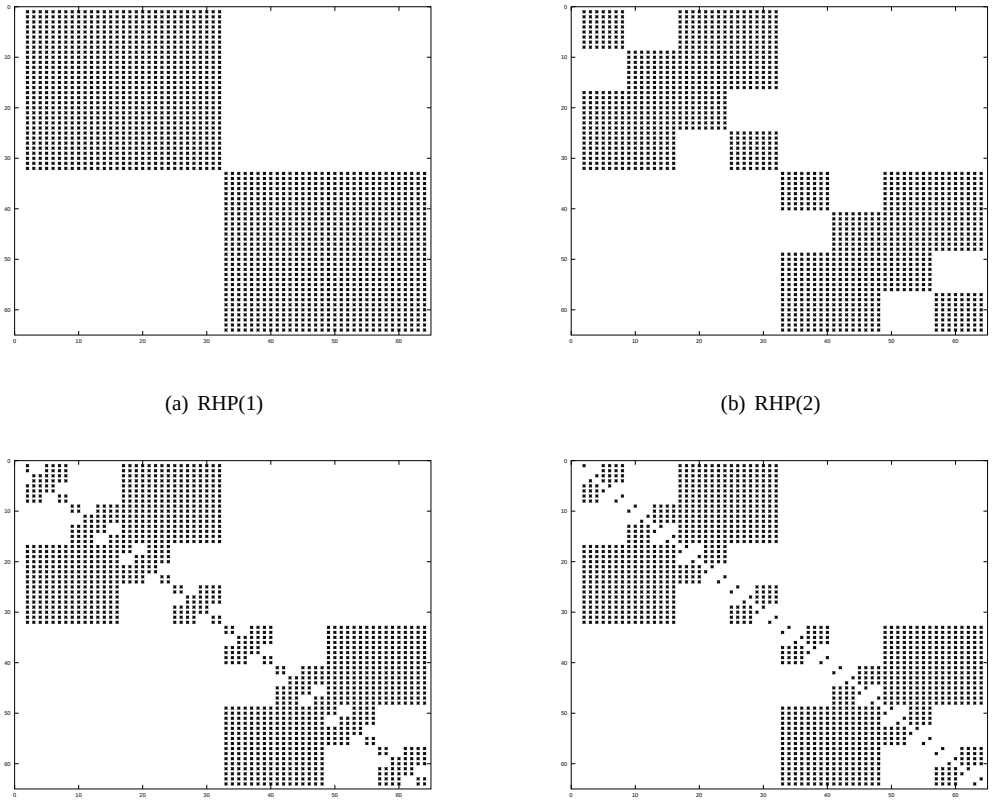
Figure 3.7: Runtimes for recursive Estimation Hessian sparsity Pattern (a) along the respective Number of Nonzeros (b) depending on the Recursion Level l for f of Listing 3.3 with $h = 14$. A Detailed View of Measurement Data is given in (c) for $h = 12$.

the proof of concept implementation of RHP described in Algorithm 3.9 and ADOL-C implementation of EHP for f of Listing 3.3. (a) shows the runtime behavior of the former for recursion levels $l = 0, 1 \ldots, 8$ for $h = 14$ with $n = 2^h = 65536$ inputs. We denote again the achieved sparsity pattern as well as the

instance of RHP for l by RHP(l). As already mentioned, RHP(l) is supposed to converge to EHP for $l = \frac{h}{2} + 1$, that is, $l = 8$ for $h = 14$. The number of nonzeros of the resulting pattern is presented in (b). A detailed view of measurement data for $h = 12$ is presented in (c), where the columns T_a and q_a present the runtime and the number of achieved colors by the application of the acyclic coloring implementation of ColPack, respectively.

We observe that the exact Hessian pattern of ∇f by RHP for both dimensions $h = 12$ and $h = 14$ is achieved at roughly the same computational time as shown in column $T(P)$ for P denoting the pattern of $\nabla^2 f$. Moreover, RHP is at least twice as fast as EHP. The reason lies in the fact that the number of performed multiplications is twice that of additions. As mentioned at the beginning of this section RHP prevent us from performing (expensive) unions of second-order dependencies as opposed to EHP as shown in line 8 of Algorithm 3.7. More substantially, the number of achieved colors of at about 4000 nonzeros denser sparsity pattern RHP(6) is equal to RHP(7). Note that the latter denotes the exact Hessian pattern.

Figure 3.3.4 shows a sparsity pattern achieved by RHP for the Hessian of f for $h = 6$. The resulting pattern are obtained in the same way as discussed in Figure 3.6 for $h = 2$.



(a) RHP(1)

(b) RHP(2)

(c) RHP(3)

(d) RHP(4)

Figure 3.8: Application of RHP for $h = 6$ resulting in $n = 64$ inputs for f. The resulting Pattern of RHP(0) is explicitly avoided as it is completely dense.

To conclude the discussion about the recursive estimation of the Hessians we have considered an ar-

tificial example function given by Listing 3.3 for illustration purposes. Our plain example clarifies the contribution of the multiplication operation to overestimation. At the same time, we used it to illustrate the difference between the complexities of the standard algorithm EHP and RHP, which recursively converges to the former. The latter was the result of exploiting the idea behind partial separability at level of elemental operation resulting in a nonlinear frontier DAG consisting of vertices corresponding to nonlinear operation of the underlying function.

We have also showed that the application of RHP for recursion level one results in the same conservative overestimated pattern as the conservative algorithm CHP [VRMN11]. Note that the latter makes use of the direct nonlinear components of the outputs. In particular, there is not really a need to build a nonlinear DAG in that case. Moreover, we have shown the efficiency of CHP on realistic problems as discussed at the beginning of this chapter. Its runtime was also presented for the computation of the Hessian pattern of another scalar function arising in context of simulated moving bed (SMB) process described at the beginning of Section 3.2.6. More substantially, coloring of the adjacency graph of the conservative Hessian pattern achieved by CHP was turned out to be much efficient in terms of achieved colors that coloring the exact Hessian pattern. We note that the latter is shown to be of quadratic complexity in worst case. A runtime comparison of both algorithms was presented in Figure 3.4.

We note that the current implementation activities of the author focus on tuning RHP to exhibit the same runtime behavior for SMB as CHP for recursion level. However, as already shown on Figure 3.6, the runtime of RHP(1) is very close to the converged version yielding the exact Hessian pattern. Hence, RHP may improve the runtime of estimating the exact Hessian pattern significantly. Moreover, further investigations focus on reducing the memory consumption of RHP to avoid running out of memory.

Moreover, as observed in column T_a of Figure 3.6 (c) the high runtime of the coloring algorithm of use prevent us so far from determining the number of colors of a particular (over) estimated sparsity pattern at reasonable time. Therefore, further investigations are planned to design sophisticated coloring algorithms [CM69], yet faster. This combined with RHP, would open up room for deeper investigations on the impact on the structure property of the concerned matrices on the resulting number of colors. Note that so far this is not really possible for large dimensions as the coloring turns out to be significantly slower than the sparsity pattern estimation.

Chapter 4

Summary and Conclusion

The main objective of Chapter 2 was on reducing the memory consumption of the reverse mode AD by application of elimination techniques on extended Jacobians of underlying functions. Here, the focus was on minimizing user's expertise in AD and the knowledge about the underlying problem F. As discussed at the beginning of the chapter the memory is an issue for almost any AD approach that accumulates derivatives such as gradients or Jacobians on an internal representation of choice kept on storage. Existing checkpointing strategies are developed face this problem for time-dependent problems. We have illustrated this for a simple example in Figure 2.1. Here, we learned that the application of checkpointing is more than an easy task.

To tackle the memory problem we have first considered the static problem of Jacobian accumulation by row elimination on extended Jacobians conceptually the same as vertex elimination on the respective DAGs. The focus there was on the analysis of the runtime and memory behavior of row elimination on extended Jacobians and their compressed row storage representations. Our numerical results have shown that Jacobian accumulation on dense extended Jacobians tends to hit the memory limit very quickly. This is not really surprising because of the quadratic (in number of rows) memory complexity of extended Jacobian. Furthermore, it has been shown that the sparsity exploitation of extended Jacobians using compressed row storage reduces the memory consumption drastically. However, we observe at the same time that Jacobian accumulation on compressed representation of extended Jacobians underperforms compared with its dense counterpart by increase in the problem size. The reason turned out to lie in the linear overhead of searching for dependencies and spots in the former. At the same time it is observed that the increase in problem size has a direct impact on the number of rows of considered matrices meaning even larger search spaces.

The impact of the latter became more clear when we have tried to parallelize the process of Jacobian accumulation in Section 2.5 using OpenMP parallel paradigm. Even though we observe much promising runtime gain by parallelization, both extended Jacobian and its compressed storage counterpart seem to hit the memory bound relatively quickly. Thus, no realistic scalability is achievable so far without facing the memory problem.

In our proof-of-concept implementation in DALG we manage to reduce the (heap) memory consumption by local application of row elimination as some kind of cross country elimination, which we presented in Section 2.6. We referred to this approach as iterative Jacobian accumulation. Our experimental results have shown that the iterative approach reduces the memory consumption drastically by application of assignment level elimination, which denotes the default iterative mode in DALG. Hence, Jacobians and gradients can be computed very cheaply in terms of memory consumption automatically. However, we have also observed that the runtime of the iterative mode on dense extended Jacobian and its compressed row representation turns out to be not as efficient as that of ADOL-C.

More precisely, the computation of the gradient of the time-dependent Heat problem (see Figure 2.34 (a)) in iterative mode is orders of magnitude slower than performing the same computation by the application of the reverse mode AD provided by ADOL-C. For this reason, assignment level elimination seems to underperform as the number of time steps increases. However, gradients of even larger dimensions can be accumulated this way, where the global reverse mode AD would fail. Thus, we consider the observed performance loss acceptable. Note that the memory reduction is black-box to the user. Nonetheless, a deeper investigation for this behavior shapes up the further direction of research on DALG. In this context, we believe that the iterative mode combined with the parallelization has the potential to perform as well.

Moreover, it might also be interesting to use graphs as a internal representation in the iterative mode. Here, it is desirable to have an efficient graph implementation both in terms of memory and data access. A first non-iterative implementation already works for small problems that we aim to extend to work in iterative and parallel modes.

In the last chapter we have introduced the constant estimation and exploitation as an alternative way to the classical sparse Jacobian and Hessian computations. Especially, in the latter coloring turns out to be the major problem to solve. Otherwise, no improvement in compression can be achieved by constant exploitation, despite the fact that retrieving constants is much more expensive than pattern estimation. In this context, it may make sense to retrieve only constant pattern in both Jacobian and Hessian cases and gain constants by computation of each case in the classical way as described by Procedures 3.3 and 3.6, respectively.

We have observed for an special problem that coloring the respective graph of the exact Hessian underperforms in terms of achieved colors compared with the conservative overestimated version. This behavior is surprising at first glance, despite the fact that the former is much sparser than the latter. Similar behavior is observed when comparing the coloring results of the exact and the variable pattern in context of constant exploitation as mentioned previously.

It looks like the traditional way of thinking ”the sparser the better” does not really hold in Hessian case. However, the reason for this behavior turned out to be rather due to the impact of the heuristics behind the coloring algorithms. Thus, implementation of more suitable heuristics as well as general characterization of ”critical” patterns are desired that will shape our further research activities.

Finally, we have introduced an algorithm for recursive estimation of the Hessian sparsity pattern and shown its convergence to the exact one on example. First runtime comparison of a proof-of-concept implementation of the recursive algorithm with that of the exact one implemented by ADOL-C has shown that the former even has the potential to improve the runtime of estimating exact Hessian pattern. However, further investigations are required to deal with the memory problem of the recursive algorithm as a DAG of nonlinear frontier components is supposed to be built at the time of evaluation the underlying function.

Bibliography

[ACM91] B. Averik, R. Carter, and J. Moré. The Minpack-2 test problem collection (preliminary version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1991.

[AGN03] A. Albrecht, P. Gottschling, and U. Naumann. Markowitz-type heuristics for computing Jacobian matrices efficiently. In *ICCS'03: Proceedings of the 2003 international conference on Computational science*, pages 575–584, Berlin, Heidelberg, 2003. Springer-Verlag.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[Bau74] F. L. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

[BBCG96] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. Proceedings Series. SIAM, Philadelphia, 1996.

[BBH+08] C. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, number 64 in LNCSE, Berlin, 2008. Springer.

[BBW04] C. H. Bischof, H. M. Bücker, and P. T. Wu. Time-parallel computation of pseudo-adjoints for a leapfrog scheme. *International Journal of High Speed Computing*, 12(1):1–27, 2004.

[BC04] M. G. Burke and R. K. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 39(4):139–154, 2004.

[BCH+06] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation – Applications, Theory, and Impelmentations*. Springer, New York, 2006.

[CC86] T. F. Coleman and J. Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7(2):221–235, 1986.

[CDK+01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CFG+02] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*. Computer and Information Science. Springer, New York, 2002.

[CG91] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*. Proceedings Series. SIAM, Philadelphia, 1991.

[CH80] M. Chein and M. Habib. The jump number of dags and posets. *Ann. Disc. Math*, 9:189–194, 1980.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[CM83] T. F. Coleman and J. J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[CP08] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.

[CPR74] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *Journal of the Institute of Mathematics and Applications*, 13:117–119, 1974.

[DER86] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[FTPR04] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software*, 30(3):266–299, 2004.

[Gay96] D. M. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 173–184. SIAM, Philadelphia, PA, 1996.

[Gil80] J.R. Gilbert. A note on the np-completeness of vertex elimination on directed graphs. *SIAM Journal on Algebraic and Discrete Methods*, 1:292, 1980.

[GJM+99] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[GLS99] W. Gropp, E. Lusk, and A. Skjelllum. *Using MPI : Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1999.

[GMP05] A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[GN02] A. Griewank and U. Naumann. Accumulating Jacobians by vertex, edge, and face elimination. In 6^e *Colloque Africain sur la Recherche en Informatique*. INRIA, 2002.

[GN03] A. Griewank and U. Naumann. Accumulating Jacobians as chained sparse matrix products. *Math. Prog.*, 3(95):555–571, 2003.

[GPW08] A. H. Gebremedhin, A. Pothen, and A. Walther. Exploiting sparsity in Jacobian computation via coloring and automatic differentiation: A case study in a simulated moving bed process. In Bischof et al. [BBH+08], pages 327–338.

[GR91] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[CG91]*, pages 126–135, 1991.

[GT82] A. Griewank and P. L. Toint. On the unconstrained optimization of partially separable functions. In Michael J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312. Academic Press, New York, NY, 1982.

[GTPW09] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther. Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS J. on Computing*, 21(2):209–223, 2009.

[GW00] A. Griewank and A. Walther. 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, mar 2000. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105. SIAM, Philadelphia, PA, 2nd edition, 2008.

[Hea97] M. T. Heath. *Scientific Computing : An Introductory Survey*. McGraw-Hill, 1997.

[Her93] K. Herley. Presentation at : Theory Institute on Combinatorial Challenges in Computational Differentiation. Mathematics and Computer Division, Argonne National Laboratory, Argonne , IL, USA, 1993.

[HNP05] L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

[HP04] L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, 2004.

[KS74] D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting Arrangements. *Inf. Process. Lett.*, 3(22):153–157, 1974.

[KW06] A. Kowarz and A. Walther. Optimal checkpointing for time-stepping procedures in ADOL-C. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 541–549, Heidelberg, 2006. Springer.

[MÖ1] F. Müller, editor. *High-Level Parallel Programming Models and Supportive Environments, 6th International Workshop, HIPS 2001 San Francisco, CA, USA, April 23, 2001, Proceedings*, volume 2026 of *Lecture Notes in Computer Science*. Springer, New York, NY, 2001.

[Mar57] H. M. Markowitz. The elimination form of the inverse and its application. *Management Science*, 3:257–269, 1957.

[MK08] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 42–53, Berlin, Heidelberg, 2008. Springer-Verlag.

[MN10] V. Mosenkis and U. Naumann. The minimum edge count problem in linearized dags. Technical report, Institute of Computer Science, RWTH Aachen University, Germany, 2010.

[Nau99] U. Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University of Dresden, December 1999.

[Nau02] U. Naumann. Elimination techniques for cheap Jacobians. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 29, pages 247–253. Springer, New York, NY, 2002.

[Nau04a] U. Naumann. Optimal accumulation of Jacobian Matrices by Elimination Methods on the dual Computational Graph. *Math. Prog.*, 99(3):399–421, April 2004.

[Nau04b] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 99(3):399–421, 2004.

[Nau06] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Prog.*, 112:427–441, 2006.

[Nau08] U. Naumann. Call tree reversal is NP-complete. In Bischof et al. [BBH+08], pages 13–22.

[Nau09] U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 7:402–410, 2009.

[Nau11] U. Naumann. *The Art of Differentiating Computer Programs*. SIAM, 2011. To appear.

[NMRC07] U. Naumann, M. Maier, J. Riehme, and B. Christianson. Automatic first- and second-order adjoints for Truncated Newton. In M. Ganzha et al., editor, *Proceedings of IMCSIT'07*, pages 541–555. PTI, 2007.

[NNH+11] S. H. K. Narayanan, B. Norris, P. Hovland, D. Nguyen, and A. H. Gebremedhin. Sparse jacobian computation using adic2 and colpack. Proceedings of International Conference on Computational Science (ICCS), to appear, 2011.

[NS11] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. Computational Science Series. Chapman & Hall / CRC Press, Taylor and Francis Group, 2011. To appear.

[Pac96] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[PT08] J. D. Pryce and E. M. Tadjouddine. Fast automatic differentiation Jacobians by compact LU factorization. *SIAM Journal on Scientific Computing*, 30(4):1659–1677, 2008.

[Pul82] W. Pulleyblank. On minimizing setups in precedence constrained scheduling. *Unpublished Manuscript*, 1982.

[Qui03] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education (ISE Editions), September 2003.

[RT78] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *J. Appl. Math.*, 34(1):176–197, January 1978.

[SG05] J. Sternberg and A. Griewank. Reduction of storage requirement by checkpointing for time-dependent optimal control problems in ODEs. In H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 99–110. Springer, New York, NY, 2005.

[Sta00] R. P. Stanley, editor. *Enumerative Combinatorics*, volume 1. Cambridge University Press, 2000.

[Sto01] J. A. Storer, editor. *An Introduction to Data Structures and Algorithms*. Birkhäuser, Boston, 2001.

[Tad08] E. M. Tadjouddine. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688, 2008.

[Tal06] E. G. Talbi. *Parallel combinatorial optimization*. Wiley-Blackwell, 2006.

[TFE98] M. Tadjouddine, C. Faure, and F. Eyssette. Sparse Jacobian computation in automatic differentiation by static program analysis. In G.Levi, editor, *Lecture Notes in Computer Science*, volume 1503 of *Static Analysis*, pages 311–326. Springer, 1998.

[TFP03] M. Tadjouddine, S. A. Forth, and J. D. Pryce. Hierarchical automatic differentiation by vertex elimination and source transformation. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 115–124. Springer, Berlin, 2003.

[UHP+09] J. Ungermann, L. Hoffmann, P. Preusse, M. Kaufmann, and M. Riese. Tomographic retrieval approach for mesoscale gravity wave observations by the PREMIER Infrared Limb-Sounder. *Atmospheric Measurement Techniques Discussions*, 2:2809–2850, 2009.

[VN07] E. Varnik and U. Naumann. Parallel Jacobian accumulation. In *Proceedings of the 2007 Conference on Parallel Computing (PARCO 2007)*, pages 311–318, September 2007.

[VNL06] E. Varnik, U. Naumann, and A. Lyons. Toward low static memory Jacobian accumulation. *WSEAS Transactions on Mathematics*, 5(7):109–117, 2006.

[VRMN11] E. Varnik, L. Razik, V. Mosenkis, and U. Naumann. Fast conservative estimation of Hessian sparsity. In M. Beckers, J. Lotz, V. Mosenkis, and U. Naumann, editors, *Fifth SIAM Workshop on Combinatorial Scientific Computing*, volume AIB-2011-09 of *Aachener Informatik Berichte*, pages 18–21. RWTH Aachen University, 2011.

[Wal08] A. Walther. Computing sparse Hessians with automatic differentiation. *ACM Transaction on Mathematical Software*, 34(1):3:1–3:15, 2008.

[Yan81] M. Yannakakist. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, March 1981.