Efficient Calculation of Jacobians Using Dynamic Programming

Uwe Naumann

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Calculation of Jacobians using Dynamic Programming

Uwe Naumann

N° 3689

Mai 1999

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Efficient Calculation of Jacobians using Dynamic Programming

Uwe Naumann *

Thème 2 — Génie logiciel
et calcul symbolique
Projet TROPICS

Rapport de recherche n° 3689 — Mai 1999 — 34 pages

Abstract: The chain rule - fundamental for Automatic Differentiation (AD) - can be applied to computational graphs representing vector functions in arbitrary orders resulting in different operations counts for the calculation of their Jacobian matrices. Very few authors have looked at this interesting subject so far and there is no generally accepted terminology for dealing with these combinations of the forward and reverse modes of AD. The minimization of the number of arithmetic operations required for the calculation of the complete Jacobian leads to a computationally hard combinatorial optimization problem.

In this paper we will describe an approach to the solution of the edge elimination problem in computational graphs that builds on the idea of optimizing chained matrix products using dynamic programming techniques. We will discuss the theory and present some test results by regarding this approach in comparison with other methods for computing Jacobians using a minimal number of arithmetic operations.

Key-words: Computational graph, edge elimination, dynamic programming

* e-mail: Uwe.Naumann@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Calcul Efficace des Jacobiennes par l'utilisation de la programmation dynamique

Résumé : Nous présentons une nouvelle approche de calcul de Jacobiennes basée sur l'élimination des arcs dans le graphe de calcul. L'objectif de cette méthode est la minimisation du nombre de multiplications scalaires nécessaires à l'accumulation de la Jacobienne d'une fonction vectorielle à un argument. Pour cela il faut résoudre un problème d'optimisation combinatoire. La méthode de la programmation dynamique offre la possibilité de résoudre ce problème.

Mots-clés : Graphe de calcul, élimination des arcs, programmation dynamique.

Contents

1 Introduction

Research in the field of **Automatic Differentiation (AD)** dates back as far as 1964, when R. E. Wengert ([Wen64]) first proposed the automation of the calculation of derivatives by a program in form of the basic forward mode. Since then a lot of work has been done in order to make AD faster and more widely applicable (see [CoGr91] and [BBCG96]). AD is a fast and convenient way for calculating directional derivatives of vector functions given as computer programs numerically up to machine precision. Notice, that AD is completely different from the approach to computing derivatives numerically through divided differences.

The computation of the Jacobian matrix of a vector function

$$F: \quad \mathbb{R}^n \supseteq D \to \mathbb{R}^m \quad : \quad \mathbf{x} \mapsto \mathbf{y} = F(\mathbf{x}) \tag{1}$$

is essential in many numerical algorithms. For practical reasons most currently available AD software packages provide only two approaches to calculating the Jacobian matrix of F at the current argument – the forward and the reverse modes which are based on the application of the chain rule to F in two different ways. However, the chain rule can be applied to computational graphs of vector functions in any arbitrary order, which leads to different operations counts for the calculation of the Jacobian matrix J. The general task of efficiently evaluating J using an approach which is sometimes referred to as *cross-country elimination* is conjectured to be NP-hard. The fact that Jacobians can be calculated by eliminating intermediate vertices in computational graphs is well known since the late 1980's. There are a few papers by various authors that motivate a closer look at this topic [GrRe91], [Bis96]. However, there is no generally accepted terminology for dealing with these combinations of the forward and reverse modes of AD, so far.

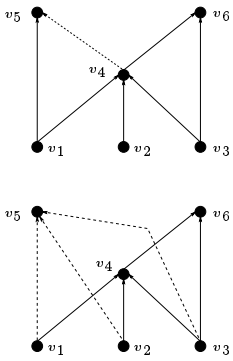

Figure 1: $(4,5)$

In our approach we expect the vector function F to be such that it can be decomposed into a sequence of functions $\varphi : \mathbb{R}^d \supseteq D_\varphi \to \mathbb{R}$ that take a vector $\mathbf{u} \in \mathbb{R}^d$ as their argument and return a value $w = \varphi(\mathbf{u})$. We call such functions **elemental**. In most cases the vector function F is given as a computer program written in some high-level programming language such as `C` or `Fortran`. This specification of F is called **evaluation routine** if it can be broken down into a sequence of scalar assignments of the form $(\mathbb{R} \ni)v_j = \varphi_j(v_i)_{i \in P_j}$ by assigning the result of every elemental function φ_j that occurs in the program to a unique intermediate variable v_j. Here P_j is the index set of the arguments of φ_j and we denote its cardinality by $|P_j|$. Since our objective is to calculate the complete Jacobian of a given vector function which consists of the partial derivatives of the m dependent variables $y_0, \ldots, y_{m-1}$ with respect to each of the n independent variables $x_0, \ldots, x_{n-1}$ it is fundamental to assume the following:

Assumption 1.1 *Given an evaluation routine of a vector function defined by Equation (1), the elemental functions φ_j $(j = 1, 2, \ldots, q+m)$ are well defined for some fixed argument*

and have jointly continuous partial derivatives

$$c_{ji} \equiv \frac{\partial}{\partial v_i}\varphi_j(v_k)_{k\in P_j} \quad \textit{for} \quad i \in P_j$$

of their respective arguments on some neighborhood $\mathcal{D}_j \subset I\!R^{n_j}$ *with* $n_j \equiv |P_j|$.

The relation between the variables in a given evaluation routine can be visualized by a directed acyclic graph $CG = (V, E)$ which contains all information needed to be able to compute the entries of the Jacobian. From now on we will refer to CG as the **computational graph** (also **c-graph**). We will distinguish between. $n \equiv |X|$ minimal [independent], $p \equiv |Z|$ intermediate and $m \equiv |Y|$ maximal [dependent] vertices:

$$X \equiv \{v_{1-n}, \ldots, v_0\}, \quad Z \equiv \{v_1, \ldots, v_p\}, \quad Y \equiv \{v_{p+1}, \ldots, v_{p+m}\}.$$

There exists a directed edge (i, j) connecting a vertex v_i with a vertex v_j in CG if v_j directly depends on v_i in F. As above we set P_j $[S_j]$ to be the set of indices of all predecessors [successors] of a vertex v_j. According to Assumption 1.1 labels c_{ji}, representing the local partial derivatives, are attached to all edges $(i, j) \in E$.

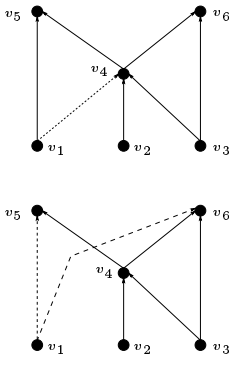

Figure 2: $(1, 4)$

We are looking for a method of transforming the c-graph of F such that we get the Jacobian J at the lowest possible cost in terms of the number of scalar multiplications involved in this process. In fact, if by successively eliminating all vertices representing intermediate variables in the underlying evaluation program (which is equivalent to eliminating all edges having either an intermediate vertex as source, or having such a vertex as target, or both, i.e. all *intermediate edges*) we get to a stage, where the c-graph represents a subgraph of the complete bipartite graph $K_{n,m}$ and the labels $c_{ji} \equiv \partial y_j/\partial x_i$ (i=0, ... ,n-1, j=0, ... ,m-1) on the edges connecting the minimal vertices with the maximal ones, are exactly the non-zero entries of the Jacobian matrix of F. The elimination of intermediate edges represents the elemental action that we will build on in our approach. It can be regarded as the chain rule applied to evaluation routines in form of c-graphs. Consequently we distinguish between two ways to eliminate an edge in CG and we will refer to them as **forward** and **backward** edge elimination.

Graphically, the forward elimination of an edge (i, j) is equivalent to connecting all predecessors of v_i with v_j (provided they have not been connected before as we do not permit multiple edges) followed by updating the existing or generating the new local partial derivatives and, finally, the deletion of (i, j). This is illustrated in Figure 1 with the help of edge $(4, 5)$. In correspondence with the chain rule we multiply the values of successive edges (i, j) and (j, k) whereas we add the values of parallel edges having the same source and the same target. Thus, the forward elimination of an edge (i, j) involves a number of scalar multiplications that is equal to the cardinality of the predecessor set of its source v_i. We will call this number the **in-degree** or **forward Markowitz degree** of (i, j) and denote it by $|P_{(i,j)}| = |P_i|$.

The graphic interpretation of the backward elimination of $(1,4)$ is shown in Figure 2. As in the case of forward elimination we simply have to insert new edges connecting v_l with all successors of v_i (if they do not exist already) and generate new or update the existing edge labels correspondingly. Finally, (l,i) is removed from the c-graph. It takes $|S_{(i,j)}|$ scalar multiplications to eliminate an edge (i,j) backward. $|S_{(i,j)}|$ denotes the **out-degree** (**backward Markowitz degree**) of (i,j) which is equal to the number $|S_j|$ of successors of the target v_j.

Let the length of a path connecting an independent vertex with a dependent one be defined as the number of edges it consists of. Then one can show that the sum of the total lengths of all distinct paths in the c-graph is strictly monotonically decreasing during the process of eliminating edges. We may conclude that edge elimination will always terminate. This seems to be obvious but it is certainly crucial for our approach.

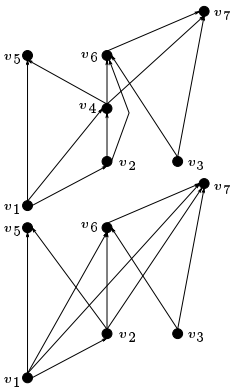

Figure 3: v_4

The forward [backward] elimination of all out-edges [in-edges] of a vertex v_i leads to the elimination of v_i itself (Figure 3). Consequently, the elimination of an intermediate vertex v_i involves $|P_i| \cdot |S_i|$ scalar multiplications. This value is usually referred to as the **Markowitz degree** of v_i [GrRe91]. In the available literature the application of the chain rule to c-graphs has been interpreted as vertex elimination. Even the few attempts to minimize the number of multiplications needed to calculate the Jacobian are based on the idea of eliminating vertices (see for example [Bis96] or [GrRe91]). However, there are problems where the optimal vertex elimination sequence does not minimize the number of multiplications. Let us illustrate this with the help of an example displayed in Figure 4. It shows a c-graph of a problem with two independent, five dependent, and two intermediate variables. There are two different vertex and numerous different edge elimination orders. The two vertex elimination orders result in $((v_i,v_k) \hat{=} 5{+}10 =)15$ and $((v_k,v_i) \hat{=} 4{+}10 =)14$ multiplications. So, using a pure vertex elimination strategy on this very simple example gives us the Jacobian for a cost of at least 14 multiplications. Now, suppose we eliminate (i,j) separately before v_k followed by the elimination of v_i. This would take only 13 multiplications which is obviously less than $14 = \min\{(v_i,v_k),(v_k,v_i)\}$ in the pure vertex elimination mode. Furthermore, we have shown that the **vertex-edge discrepancy** does not exceed a factor of $(\sqrt{2}(2-\sqrt{2}))^{-1}$ for c-graphs containing two intermediate vertices. The problem remains unsolved for c-graphs with $p > 2$ intermediate vertices. However, building on numerous test results we conjecture that the vertex-edge discrepancy will be less or equal to 2 for the general case.

Given an evaluation procedure of a vector function F we are going to think about a way to compute its Jacobian at a minimal cost. For reasons described in [Nau99b] we have decided to regard the number of multiplications required to compute the complete Jacobian as the cost and we will denote this objective function by $\mathbf{Cost}\{J\}$. Specifically, we will take the c-graph CG of F and we will search for an edge elimination sequence that minimizes $\mathbf{Cost}\{J\}$.

The successive elimination of edges from the c-graph defines the so-called **metagraph**

$$M = M(CG) = (V_M, E_M)$$

the vertices $w_k \in M$ of which represent all different c-graphs that could possibly be derived from CG by edge elimination. Labeling the edges in M with the cost of getting from the graph represented by their source to the one associated with their target we end up with a shortest path problem on the metagraph. The edge labels are considered to be the distances and we will refer to this problem as the **general edge elimination problem**. The difficulties arise from the fact that both the number of vertices and the number of edges in the metagraph grow exponentially with the number of intermediate vertices in the original c-graph. Therefore, an exhaustive search as well as any algorithm for computing a shortest path in M are not practicable. In order to decrease the complexity of the problem to solve we have to make certain restrictions, thus reducing both size of the metagraph defined by the number of its stages and the number of different paths to check. This approach will lead to subgraphs which are then subject to an analogous shortest path problem. With every restriction ρ we can associate a corresponding minimal cost $\mathbf{Cost}_\rho\{J\}$ of computing the Jacobian by solving the shortest path problem on the induced subgraph $M_\rho \subseteq M$ of the metagraph, i.e. $\mathbf{Cost}_\rho\{J\} = c_\rho \cdot \mathbf{Cost}\{J\}$ for some $c_\rho \geq 1$. Obviously, we would like the factor to be as small as possible as for large values c_ρ the chosen restriction ρ would not be very useful. One possibility is the restriction to eliminating vertices in the c-graph which will be exploited in this paper. This approach leads to the so-called **vertex metagraph** $M_\mathbf{V} \subseteq M$ an instance of which is shown in Figure 5 for a c-graph containing 4 intermediate vertices. For further information on the general edge elimination problem refer to [Nau99b].

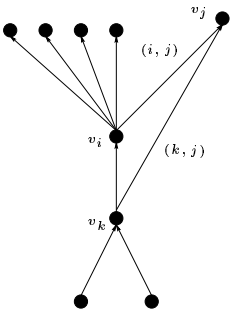

Figure 4: Surprise

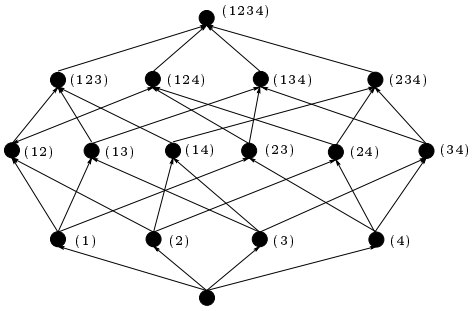

Figure 5: Vertex Metagraph $M_\mathbf{V}$

We have developed new and adapted some well-known algorithms for solving the combinatorial optimization problem. These methods contain local heuristics as well as a collection of simulated annealing schedules and dynamic programming algorithms for the computation of nearly optimal vertex elimination sequences. The latter will be looked at more closely in this paper. All the approaches proposed are implemented as parts of a program named **OESCOMP** [Nau99a], which all our test results will be based on.

Considering local heuristics for edge or vertex elimination in c-graphs there are, obviously, two trivial cases: Let $v_i < v_j$ denote the situation where a vertex v_i is eliminated before v_j. The elimination order yielding $\forall w_k \in M : \; v_i < v_j \Leftrightarrow i < j$ will be referred to as the **forward vertex elimination mode (V_F)**. Analogous, we define the **backward vertex elimination mode (V_B)**. Both **V_F** and **V_B** are considered to be equivalent to

the corresponding edge elimination modes. Since our objective is to minimize the cost of computing the complete Jacobian it certainly makes sense to think about how cheaply a particular intermediate vertex v_i can possibly be eliminated. The logical result would be to order all intermediate vertices increasingly by their Markowitz degrees at each stage of the metagraph. At a particular stage $w_k \in M_{\mathbf{V}}$ we eliminate the vertex with the lowest Markowitz degree among all intermediate vertices. This approach is known as the **Lowest-Markowitz-Degree-First (V_LM)** strategy:

$$w_k \in M_{\mathbf{V}}: \quad v_i < v_j \quad \Leftrightarrow \quad |P_i|_k \cdot |S_i|_k \ \leq \ |P_j|_k \cdot |S_j|_k.$$

Here $|P_i|_k$ [$|S_i|_k$] denotes the number of predecessors [successors] of a vertex $v_i \in CG$ at a certain stage $w_k \in M_{\mathbf{V}}$ in the (vertex) metagraph. As a well-known heuristic for the minimization of the generated fill-in during the solution of sparse systems of linear equations the Markowitz based approach to the vertex elimination problem in c-graphs has already been examined in several papers like for instance in [GrRe91]. However, numerous tests showed that, in general, **V_LM** does not deliver optimal elimination sequences.

Let us consider an improved version of the **V_LM** heuristic. We define the **input-dependency degree** of a vertex $v_j \in Y \cup Z$ as $\mathtt{id}_j = k$ if there are paths connecting a maximum of k minimal vertices with v_j. Analogous the **output-dependency degree** of a vertex $v_j \in X \cup Z$ is $\mathtt{od}_j = k$ if a maximum of k maximal vertices are reachable from v_j. For a vertex $v_j \in Z$ we define its **dependency degree** as $\mathtt{dd}_j \equiv \mathtt{id}_j \cdot \mathtt{od}_j$. The dependency degree of a vertex is invariant with respect to the pure vertex elimination strategy which makes it ideal for the implementation of new heuristics. In particular, we have implemented the **Lowest-Relative-Markowitz-Degree-First (V_LR)** heuristic:

$$w_k \in M_{\mathbf{V}}: \quad v_i < v_j \quad \Leftrightarrow \quad |P_i|_k \cdot |S_i|_k - \mathtt{dd}_i \ \leq \ |P_j|_k \cdot |S_j|_k - \mathtt{dd}_j.$$

Numerous tests showed that this heuristic is superior to **V_LM** in most cases. We will illustrate this using the example c-graph displayed in Figure 6. Obviously, it is easy to solve the resulting vertex elimination problem in three intermediate variables. It is even possible to check all different orders. Notice that **V_LM** does not deliver the minimum operations count in this very simple case. It behaves like **V_F** resulting in 22 multiplications. Considering **V_LR** we have $\mathtt{dd}_1 = 6$, $\mathtt{dd}_2 = \mathtt{dd}_3 = 9$ and since $6 - 9 < 4 - 6$ the vertex v_2 is eliminated first. At the next stage we get $6 - 9 < 8 - 6$ and therefore **V_LR** would act the same way as **V_B** delivering the minimal operations count which is 18 for this example.

Considering strategies for edge elimination we have, among others, proposed the class of fill-in based heuristics. The fill-in generated by the elimination of an intermediate edge (i, j) is defined as the number of edges in the c-graph after minus the number of edges before the elimination of (i, j). Every edge labeled with an elementary partial derivative is a potential factor in the elimination process which represents the computational process of accumulating the complete Jacobian. A logical consequence of these observations is to keep the number of these factors as low as possible which implies the minimization of the locally produced fill-in. This idea is exploited in the **Lowest-Fill-in-First** edge elimination heuristic (**E_LF**). Here, we always eliminate the edge producing the lowest fill-in next.

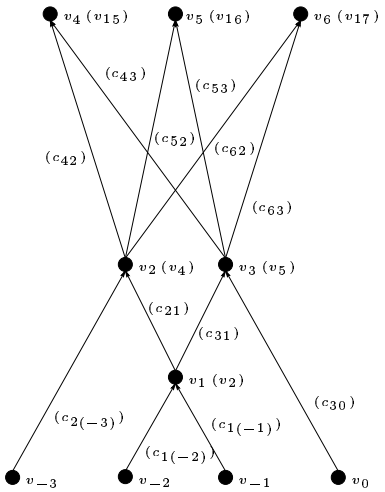

Figure 6: CG

A different approach to solving the general edge elimination problems is built on the idea of simulated annealing. Simulated annealing is a global optimization method that distinguishes between different local optima. Starting from an initial point, the algorithm takes a step and the function is evaluated. When minimizing a function, any downhill step is accepted and the process repeats from this new point. An uphill step may be accepted. Thus, the algorithm can escape from local optima. This uphill decision is made by the Metropolis criterion. As the optimization process proceeds, the length of the steps decline and the algorithm closes in on the global optimum. Since the algorithm makes very few assumptions regarding the function to be optimized, it is quite robust with respect to combinatorial problems. The degree of robustness can be adjusted by the user. In [Nau99c] we have discussed several annealing schedules for solving the vertex elimination problem in c-graphs. These could easily be adapted to the more general edge elimination problem. For comparing the results achieved by the dynamic programming approach with the ones delivered by other methods we will denote the most consistent schedule proposed in [Nau99c] by **SA**.

Edge elimination in c-graphs makes full use of the structural sparsity of the given problem. It is often possible to reduce the number of multiplications required to accumulate the complete Jacobian by a factor of three and more compared to the method proposed by Newsam and Ramsdell [NeRa83]. The savings compared to the dense forward and reverse modes are even more significant. When presenting test results in Section 4 we will consider the achieved values relative to lower bounds for values of the best choice out of dense forward and reverse modes (**DM**), i.e.

$$\mathbf{Cost_{DM}}\{J\} = \min\{n(m+p), m(n+p)\},$$

and the corresponding minimum operations count achieved by a uni-directional approach of the method by Newsam and Ramsdell (**NR**) which is

$$\mathbf{Cost_{NR}}\{J\} = \min\{\hat{n}(m+p), \hat{m}(n+p)\}.$$

$\hat{n}$ and $\hat{m}$ denote the maximal number of non-zero elements per row and column of the Jacobian, respectively.

2 Dynamic Programming

Dynamic programming techniques are very useful for the solution of problems that can be partitioned into subproblems the size of which can be kept small. They are based on the recursive solution of each of the subproblems in order to construct the solution of the composite problem. If the sum of the sizes of the p subproblems is a multiple $c \cdot p$ of p for some constant $c > 1$ the dynamic programming algorithm is likely to be polynomial in time complexity [AHU74]. However, if the division of a problem of size p delivers p subproblems of size $p - 1$ then we can expect the recursive algorithm to have exponential growth. An example for this case is sketched in [Bis96].

The two key ingredients that an optimization problem must have for dynamic programming to be applicable are referred to as **optimal substructure** and **overlapping subproblems**. A problem exhibits optimal substructure if an optimal solution to the problem is built on optimal solutions to subproblems. Furthermore, the space of these subproblems must be small in the sense that a recursive algorithm for the problem repeatedly solves the same subproblems, rather than always generating new instances. Ideally, the number of distinct subproblems grows polynomially with the input size of the composite problem. Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once. The results are stored in a table where they can be looked up when needed, using constant time.

2.1 Chained Matrix Products

When dynamic programming is introduced in the literature the authors often use the so-called *chained matrix multiplication problem* as an introductory example [AHU74]. Since the context in which we are going to use dynamic programming is very close to this problem we decided to have a closer look at its classical formulation. Consider the product of p matrices

$$A = A_1 \cdot A_2 \cdot \ldots \cdot A_p \tag{2}$$

where each A_i has r_{i-1} rows and r_i columns. The number of multiplications required to compute A depends strongly on the order in which the matrices A_i are multiplied together. Checking all possible orders is an exponential process. However, dynamic programming provides an $O(p^3)$ algorithm.
Matrix multiplication is associative and so all parenthesizations yield the same product. We call a product of matrices **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. We denote the number of different parenthesizations of a product of k matrices by $\Pi(k)$. Obviously, we can split the above matrix product between the l-th and the $(l+1)$-st factor into two subchains and parenthesize them independently. Thus, we get the recurrence

$$\Pi(k) \quad = \quad \begin{cases} 1 & \text{if } k = 1 \\ \sum_{l=1}^{k-1} \Pi(l)\Pi(k-l) & \text{if } k \geq 2 \end{cases}, \tag{3}$$

the solution of which is exactly the sequence of **Catalan numbers** $\Pi(k) = \mathcal{C}(k-1)$ where

$$\mathcal{C}(k) = \frac{1}{k+1}\binom{2k}{k}.$$

So, the number of different parenthesizations of a product of k matrices is exponential in k which makes a naive exhaustive search algorithm inapplicable even for not very large values of k. But what is the difference that dynamic programming could possibly make?

Suppose that we split the chained matrix product given by Equation (2) between the l-th and the $(l+1)$-st factor into the two subproducts

$$A_{\texttt{left}} = A_1 \cdot A_2 \cdot \ldots \cdot A_l \quad \text{and} \quad A_{\texttt{right}} = A_{l+1} \cdot A_{l+2} \cdot \ldots \cdot A_p.$$

Consider an optimal parenthesization. The key observation is that the parenthesization of $A_{\texttt{left}}$ within an optimal parenthesization of A must be optimal too, as if there was a cheaper way to parenthesize $A_{\texttt{left}}$, substituting it in the optimal parenthesization of A would result in a less costly parenthesization of A. Similar observations yield the same contradiction for $A_{\texttt{right}}$. Thus, an optimal solution to the chained matrix product problem consists of optimal solutions to subproblems. This is what we call optimal substructure representing one of the prerequisites for successfully implementing a dynamic programming algorithm which is polynomial in the input size of the problem. The second ingredient – the overlapping subproblems property – can be explained with the help of Figures 7 and 8. Consider the computation of an optimal parenthesization of a product of four matrices $A_1, \ldots, A_4$. With Equation (3) we have that there are five alternative ways to exploit the associativity of the chained matrix product. The sink (1234) in the table look up graph displayed in Figure 7 corresponds to the fully parenthesized product of the four matrices. We denote the optimal parenthesized matrix product $A_{i_1} \cdot \ldots \cdot A_{i_k}$ by $(i_1, \ldots, i_k)$. A chained matrix product containing k factors can be split into two subchains in exactly $k-1$ ways. By the *optimal substructure criterion* each of the resulting subproblems has to be optimal with respect to its parenthesization. Therefore, the solution of this final subproblem with $k = 4$ factors involves the solutions of $2(k-1) = 6$ subproblems. It makes use of the earlier computed solutions of the two trivial subproblems (1) and (4) and of the four subproblems (12), (34), (123), and (234). Furthermore, as illustrated by the upper small picture in Figure 7 the solution to the subproblem (234) is also based on the solution of (34). So, a naive recursive algorithm would have to solve the subproblem (34) twice in order to find an optimal parenthesization for (1234) (analogous (12) would be solved twice). The idea behind the corresponding dynamic programming algorithm is to avoid the repeated solution of these *overlapping subproblems* by solving each of them once and storing the result in a table which has precisely the same structure as the table look up graph.

The above observations lead to the following dynamic programming algorithm for solving the classical chained matrix product. Let m_{ij} be the minimal number of multiplications required for computing $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ for $1 \leq i \leq j \leq p$. Obviously,

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m_{ik} + m_{(k+1)j} + r_{i-1} r_k r_j\} & \text{if } j > i \end{cases}$$

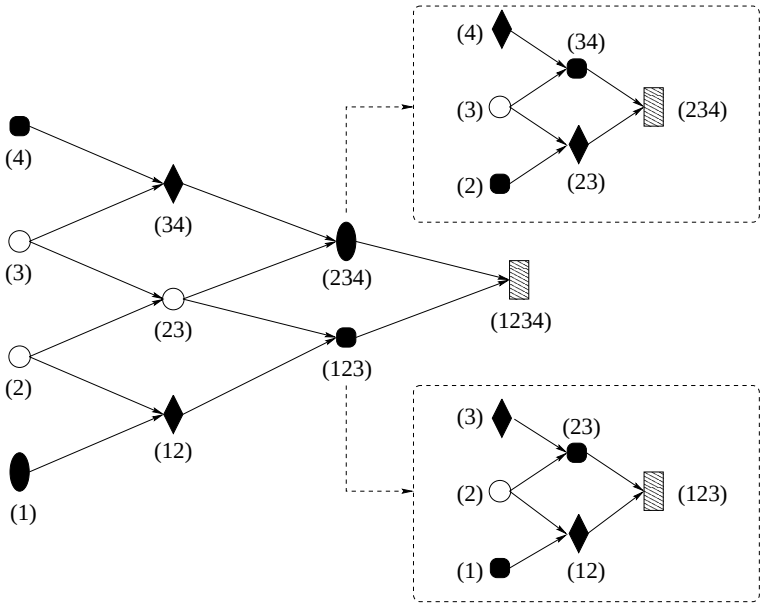

Figure 7: Table Look Up Graph

where $r_{i-1}r_k r_j$ is the cost of multiplying $A_{(k+1)j}$ by A_{ik}, for

$$A_{(k+1)j} = A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_j \quad \text{and} \quad A_{ik} = A_i \cdot A_{i+1} \cdot \ldots \cdot A_k.$$

The corresponding dynamic programming algorithm calculates the m_{ij} in the order of increasing difference in the subscripts.

The table look up graph T associated with the chained matrix multiplication problem has some special properties. First, we observe that

$$\forall v_i \in T \; : \; 1 \leq \{|P_i|, |S_i|\} \leq 2$$

which is due to the associativity of the matrix multiplication. T can be divided into *stages* with indices $0, 1, \ldots, k-1$. There are exactly k stages for a matrix chaining problem involving k factors resulting in a table look up graph which contains

$$\sum_{i=1}^{k} i \quad = \quad \binom{k+1}{2} \quad = \quad \frac{1}{2}\left(k^2 + k\right) \quad = \quad |V_T|$$

vertices and

$$2 \cdot \sum_{i=1}^{k-1} i \quad = \quad 2 \cdot \binom{k}{2} \quad = \quad k^2 - k \quad = \quad |E_T|$$

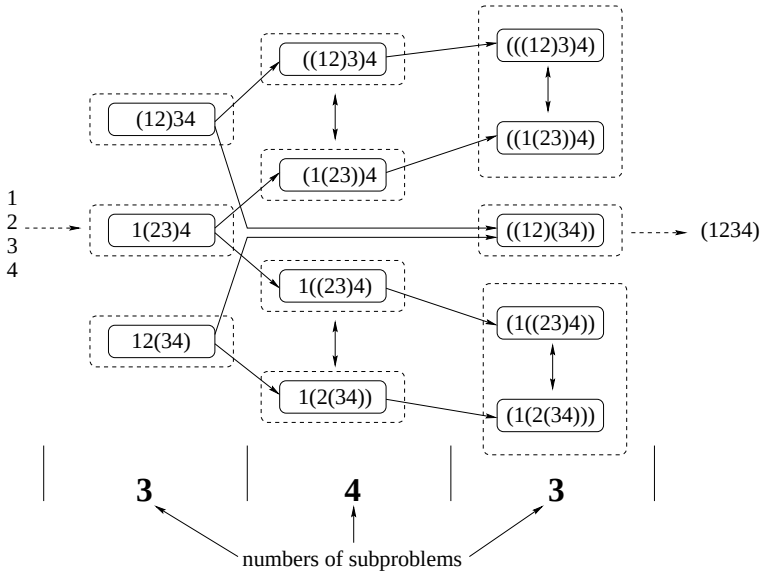

Figure 8: Metagraph

edges. Obviously, the number of how often the solution of a given subproblem at stage j in the table look up graph is involved in the solution of a problem at stage $i > j$ is either zero, if there is no path in T connecting the vertices corresponding to the two subproblems, or it is $i - j$ which is the length of any path between the two subproblems in T. With the number of subproblems at stage i $(i = 0, \ldots, k-1)$ in T being equal to $k - i$ and the solution of all problems at this stage involving i subproblems we end up with a total number of

$$\sum_{i=1}^{k-1}(k-i)i$$

subproblems to be solved. In Figure 8 we have illustrated this fact by listing the numbers of subproblems corresponding to the different stages in the table look up graph. We end up with an algorithm which is cubic in the number k of factors in the chained matrix product.

2.2 Classification

So far, we have assumed the derivative objects associated with the vertices in the c-graph to be scalars. In general, we will distinguish between different *types* of c-graphs, depending on whether the values associated with each of their vertices are scalars or vectors. Moreover, we will look at different *modes* of calculating directional derivatives based on the number of tangents [gradients] that are computed by one forward [reverse] sweep through the c-graph.

Definition 2.1 *A c-graph CG is called* **vector c-graph** *if the elemental functions Φ_j associated with each its vertices v_j are vector valued, i.e. if*

$$\Phi_j : I\!R^{d_j^{\mathtt{pred}}} \supseteq D \to I\!R^{d_j} \qquad for \quad j = 1, \ldots, p+m$$

and with $d_j^{\mathtt{pred}} = \sum_{i \in P_j} d_i$. If $d_j = 1$ for all $j = 1 \ldots, p+m$ then CG is called **scalar c-graph** *and the elemental functions are denoted by φ_j.*

Derivative vectors $\tilde{\mathbf{v}}_j \in I\!R^{|V|}$ are associated with every vertex of a vector c-graph.

Definition 2.2 *A c-graph CG is said to be regarded in the context of* **vector mode** *of AD if during one (forward, reverse, or cross-country) sweep the local partial derivatives labeling the edges of CG are applied to the $q \geq 1$ components of a vector simultaneously. We speak of the* **scalar mode** *if $q = 1$.*

Building on the above definitions we introduce the classification shown in Figure 9. The *scalar mode on scalar c-graphs* represents the simplest form of looking at the propagation of directional derivatives in a c-graph of a given vector function. The extension to bundles of tangents and gradients leads to the *vector mode on scalar c-graphs* which is usually referred to as the vector mode of AD.
There is no need to introduce the *scalar mode on vector c-graphs* separately as it is useful to regard it as a special case of the *vector mode on vector c-graphs*, which we will also refer to as the **general vector mode**. In the general vector mode we allow the elemental functions obtained by recording the evaluation trace to be vector-valued. Recapitulate that, so far, we have always decomposed the evaluation program into a sequence of scalar assignments of the form $v_j = \varphi_j(v_i)_{i \in P_j}$. By Assumption 1.1 and using the well-known analytical expressions for the partial derivatives of the intrinsic functions provided by most high-level programming languages this allows the straight forward computation of the values of the local partial derivatives of any $\varphi_j : I\!R^{|P_j|} \supseteq D \to I\!R$ for $j = 1, \ldots, p+m$ with respect to each of their arguments v_i $(i \in P_j)$ for the given value. Classic AD exploits this information for the computation of directional derivatives by forward [backward] propagating tangents [gradients] or the corresponding bundles in vector mode. Suppose we extend the term *elemental function* such that we do not necessarily restrict its range to $I\!R$ but allow *elemental* assignments to vectors as unique intermediate variables. This enables us to process more complex evaluation programs efficiently by, for example, regarding subroutines as elementals as exploited in the *hierarchical elimination* approach. Here we simply structure large-scale problems hierarchically which represents the only practicable way to apply the cross-country elimination method to problems resulting in c-graphs which consist of several ten thousand intermediate vertices. The c-graphs of most of these problems may not even fit into the memory of the available computer systems. So, a hierarchical treatment will make sense even if the chain rule is not applied in the cross-country mode but in the basic forward and reverse modes of AD.
In general vector mode a single assignment code is given by

$$\mathbf{v}_j = \Phi_j(\mathbf{v}_i)_{i \in P_j} \quad \text{where} \quad \Phi_j : I\!R^{|P_j|} \supseteq D \to I\!R^{d_j} \quad \text{for} \quad j = 1, \ldots, p+m$$

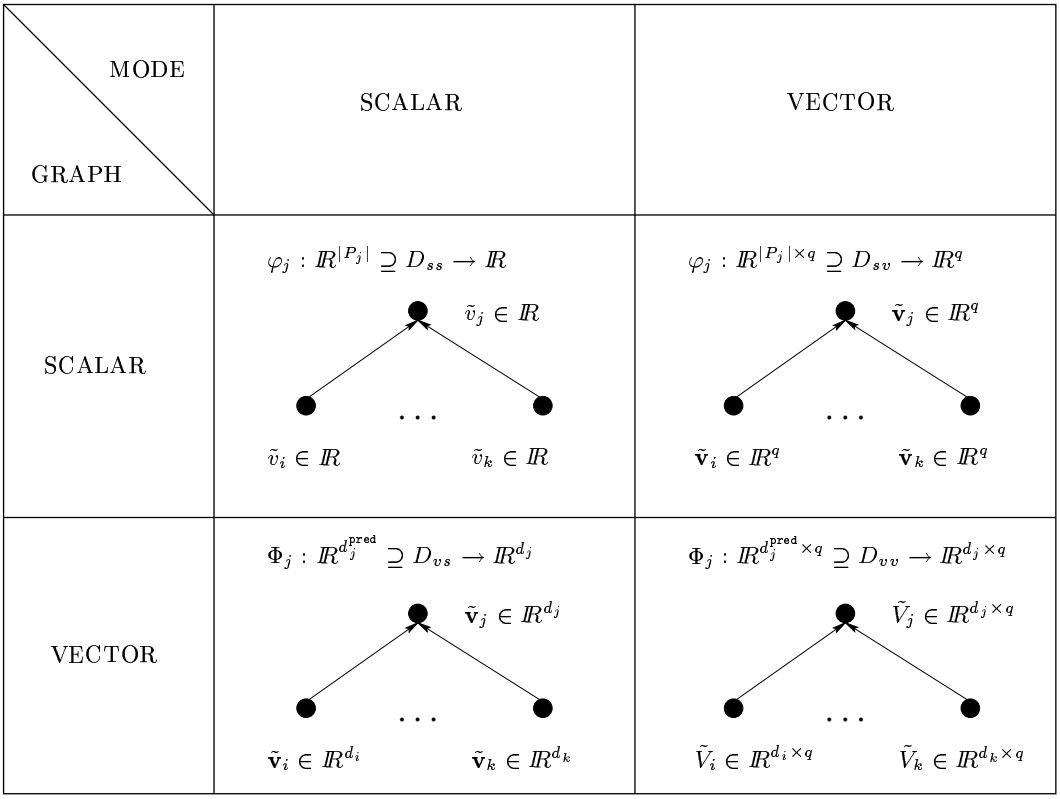

Figure 9: Classification

and the local partial derivatives become local Jacobian matrices

$$C_{jk} = \frac{\partial}{\partial \mathbf{v}_k} \Phi_j(\mathbf{v}_i)_{i \in P_j} \in I\!R^{d_j \times d_k} \quad \text{with} \quad j \in \{1, \ldots, p+m\} \quad \text{and} \quad k \in \{1-n, \ldots, p\}.$$

Analogous to the classic vector version of AD we can write down expressions for the forward and reverse modes. Again, we will propagate bundles of $q \geq 1$ tangents [gradients] forward [backward].

General forward vector mode (non-incremental form):

$$(I\!R^{d_k \times q} \supseteq) \ \dot{V}_k = \sum_{j \in P_k} C_{kj} \dot{V}_j \qquad \text{for} \quad k = 1, \ldots, p+m.$$

General reverse vector mode (incremental form):

$$(I\!R^{q\times d_j} \supseteq)\ \bar{V}_j \mathrel{+}= \bar{V}_k C_{kj} \qquad \text{for} \quad j \in P_k \quad \text{and} \quad k = p+m,\ldots,1.$$

Expressions for the two remaining, but for reasons described in [Nau99b] not very practicable modes can be derived easily. Notice that in general reverse mode we interpret the $\bar{V}_j$ as bundles of q row vectors. Let us have a closer look at the elimination of edges in vector c-graphs. For $\mathbf{v}_i \in I\!R^{d_i}$, $\mathbf{v}_j \in I\!R^{d_j}$, and $\mathbf{v}_k \in I\!R^{d_k}$ and with both (i,j) and (j,k) in CG we get local Jacobians $C_{ji} \in I\!R^{d_j \times d_i}$ and $C_{kj} \in I\!R^{d_k \times d_j}$. The forward [backward] elimination of all out-edges [in-edges] of a given intermediate vertex $v_j \in CG$ is equivalent to eliminating v_j itself. Since the edge labels are matrices the elimination of edges involves matrix products. Suppose we eliminate (j,k) forward. Then we get $C_{ki} \mathrel{+}= C_{kj} \cdot C_{ji}$ for all $i \in P_j$ using the earlier introduced `C`-style notation. As in scalar mode we connect all predecessors of v_j with v_k and perform the corresponding matrix multiplications to calculate the labels of the inserted edges. If an edge from v_i to v_k already exists then we additionally have to build the corresponding sum of the two matrices. Thus, we get the following number of multiplications involved in the forward elimination of an edge (j,k) in the general vector mode:

$$\mathbf{OPS_F}\,\{(j,k)\} = d_k d_j \sum_{i\in P_j} d_i.$$

As stated in Section 1, we will always count multiplications as a complexity measure, ignoring the number of additions involved in the calculation. Especially in this case where $d_k d_j \sum_{i\in P_j} d_i$ scalar multiplications are performed compared to at most $d_k \sum_{i\in P_j} d_i$ additions this approach seems to be appropriate as the latter are surely dominated by the former in terms of the time required to run them.

The backward elimination of (i,j) is performed correspondingly, i.e. $C_{ki} \mathrel{+}= C_{kj} \cdot C_{ji}$ for all $k \in S_j$. Again we connect v_i with all successors of v_j and perform the corresponding matrix multiplications possibly followed by additions. We get the following number of multiplications needed for the backward elimination of an edge (i,j):

$$\mathbf{OPS_B}\,\{(i,j)\} = d_i d_j \sum_{k\in S_j} d_k.$$

In consistency with Section 1 we call $\mathbf{OPS_F}\,\{(i,j)\}$ [$\mathbf{OPS_B}\,\{(i,j)\}$] the **forward [backward] Markowitz degree** of an edge (i,j) in general vector mode.

Vertex elimination: The elimination of a vertex v_j can be regarded as the forward elimination of all of its out-edges or, equivalently, as the backward elimination of its in-edges. In either case we get for the number of multiplications involved

$$\mathbf{OPS}\,\{v_j\} = d_j \left(\sum_{i\in P_j} d_i\right)\left(\sum_{k\in S_j} d_k\right)$$

which reduces to the **Markowitz degree** of v_j in the scalar mode if $d_j = 1$ for all j.

2.3 Jacobians as Chained Matrix Products

Before we start our discussion of the representation of Jacobians as chained matrix products it is necessary to give a few definitions and to introduce the notation.

Definition 2.3 *Let $CG = (V, E)$ be a c-graph. A numbering*

$$\mathcal{I}: \quad V(CG) \to \{1-n, \ldots, p+m\}$$

of the vertices in CG is called a **consistent numbering** *(also* **consistent indexing scheme***) if it induces a topological order of the vertices of CG with respect to dependency, i.e.*

$$v_i \prec^* v_j \quad \Rightarrow \quad \mathcal{I}(v_i) < \mathcal{I}(v_j).$$

The **extended local Jacobians** C_i are defined for $i = 1, \ldots, p+m$ as

$$C_i \equiv \begin{pmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ c_{i(1-n)} & \cdots & c_{i(i-n)} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{pmatrix} \in I\!R^{|V| \times |V|}$$

where the c_{ij} are the elementary partial derivatives of $v_i = \varphi_i(\{v_j : j \in P_i\})$ with respect to its predecessors occurring in the $(n+i)$-th row of C_i. Thus, we have that $c_{ij} = 0$ if $j \notin P_i$. We define the extended local Jacobian of a set of vertices with indices in $N \subseteq Z \cup Y$ as

$$C_N \equiv \sum_{i \in N} C_i.$$

Analogous, we define the **adjoint extended local Jacobians** $\bar{C}_i$ for each of the intermediate or minimal vertices $v_i \in X \cup Z$, i.e. we set for $i = 1-n, \ldots, p$

$$\bar{C}_i \equiv \begin{pmatrix} 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & c_{(i+1)i} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & c_{(p+m)i} & 0 & \cdots & 0 \end{pmatrix} \in I\!R^{|V| \times |V|}$$

where the c_{ji} are the local partial derivatives of the successors of v_i with respect to v_i itself occurring in the $(n+i)$-th column of $\bar{C}_i$. Again, the extended adjoint local Jacobian of a vertex set $\{v_i : i \in N\}$, with $N \subseteq X \cup Z$, is defined as

$$\bar{C}_N \equiv \sum_{i \in N} \bar{C}_i.$$

In order to simplify the notation used for the further argumentation it is useful to define projections of vectors and matrices to certain components, rows or columns:

$$I\!R^{|V| \times |V|} \ni \quad I(M) \equiv \sum_{i \in M} \mathbf{e}_i \cdot \mathbf{e}_i^T$$

for a given index set $M \subseteq \{0, \ldots, |V|-1\}$ and with $\mathbf{e}_i \in I\!R^{|V|}$ denoting the i-th Cartesian basis vector. Then we have that $I(M) \cdot A$ maps A to the rows indices of which are elements of M and, analogous, $A \cdot I(M)$ extracts the corresponding columns from A. For a given vector $\mathbf{a} \in I\!R^{|V|}$ $I(M) \cdot \mathbf{a}$ sets all those components of $\mathbf{a}$ indices of which are not in M to zero.

With the terminology introduced above there is a representation of the complete Jacobian J as a chained matrix product with factors representing the local extended Jacobians associated with each single intermediate variable given by

$$J = Q_m(C_{p+m} + I - \mathbf{e}_{p+m}\mathbf{e}_{p+m}^T) \ldots (C_1 + I - \mathbf{e}_1\mathbf{e}_1^T)P_n^T. \tag{4}$$

There is an alternative way of approaching the same problem in terms of the local extended adjoint Jacobians giving us a second version of expressing the Jacobian:

$$J = Q_m(\bar{C}_p + I - \mathbf{e}_p\mathbf{e}_p^T) \ldots (\bar{C}_{1-n} + I - \mathbf{e}_{1-n}\mathbf{e}_{1-n}^T)P_n^T. \tag{5}$$

As before, P_n and Q_m are the matrices that project a given vector of length $n+p+m$ to its first n and its last m components, respectively. With the help of P_n and Q_m we can extract the complete Jacobian matrix from the extended Jacobian J_e which is actually computed by the above matrix products. By introducing the notation

$$B_j = C_j + I - \mathbf{e}_j\mathbf{e}_j^T \quad \text{and} \quad \bar{B}_j = \bar{C}_j + I - \mathbf{e}_j\mathbf{e}_j^T \tag{6}$$

we get

$$J = Q_m(B_{p+m} \ldots B_1)P_n^T \qquad \textbf{scalar tangent chain,} \tag{7}$$

$$J = Q_m(\bar{B}_p \ldots \bar{B}_{1-n})P_n^T \qquad \textbf{scalar adjoint chain.} \tag{8}$$

Regarding the C_j's as *additive* local extended Jacobians we could call the B_j's *multiplicative* local extended Jacobians. The C_j's turned out to be advantageous for deriving rules for vertex elimination in c-graphs. However, since there is no substantial difference in what both B_j and C_j are supposed to express we will not distinguish between them in our terminology and refer to both of them as local extended Jacobians. Analogous, this applies to the adjoint version.

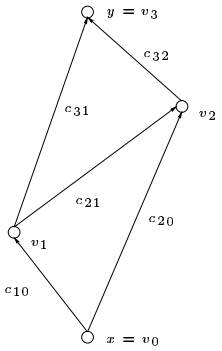

Figure 10: B_2B_1 (1)

Now we ask for a parenthesization which minimizes the number of multiplications performed while computing the above chained matrix products. We could adapt the dynamic programming algorithm for the solution of the *chained matrix multiplication problem* and would thus get a solution in a time which is polynomial in the length of the matrix chain, i.e. polynomial in the number of intermediate vertices in the c-graph. Representing a restriction on the metagraph this is not exactly the problem that we actually want to solve. The local extended [adjoint] Jacobians are not static during the elimination process. They may be changed by the elimination of one single edge. So, we have to update them after each matrix multiplication. This makes the whole problem much more complicated. Nevertheless, we have implemented the chained matrix product optimization algorithm and it turned out to deliver very promising results in many cases. Therefore, we will present a detailed discussion of this dynamic programming approach.

The calculation of either one of the scalar chained matrix products is equivalent to some edge elimination strategy, thus, making it a method for approximately solving the shortest path problem on the corresponding subgraph of the metagraph. The product of two elemental extended [adjoint] Jacobians is equivalent to the elimination of the edges connecting the two corresponding vertices in CG. In particular

$$\begin{aligned} B_{[j]} \equiv B_j B_i \quad &\hat{=} \quad \text{forward elimination of } (i,j) \\ \text{and} \quad \bar{B}_{[j]} \equiv \bar{B}_j \bar{B}_i \quad &\hat{=} \quad \text{backward elimination of } (i,j). \end{aligned}$$

$B_{[j]}$ $[\bar{B}_{[j]}]$ denotes the local extended [adjoint] Jacobian of a **supernode** which is a set of vertices belonging to one and the same class induced by the relation $\sim$. We have

$$i \sim j \quad \Leftrightarrow \quad i \not\prec^* j \quad \text{and} \quad j \not\prec^* i. \tag{9}$$

The chained matrix product approach is closely related to the elimination of edges in vector mode. We observe that the calculation of

$$(B_{j_k} \ldots B_{j_1})(B_{i_l} \ldots B_{i_1}) \quad \equiv \quad B_{[j_k]} B_{[i_l]}$$

is equivalent to the forward elimination of the edge $([i_l],[j_k])$ in vector mode where $\dot{\mathbf{v}}_i \in I\!R^l$ and $\dot{\mathbf{v}}_j \in I\!R^k$.

Summarizing the restrictions on the metagraph caused by the application of a scalar chained matrix product optimization method we get:

1. We do not allow products of the form $(B_{j_k} \ldots B_{j_1})(B_{i_l} \ldots B_{i_1})$ where $j_1 - i_l > 1$. This means that there are numerous edges which cannot be eliminated at the scalar level in the sense of the classification introduced in Section 2.2. The number of edges which could possibly be eliminated at this level is at most $p+m$ for the tangent chain and $p+n$ for the adjoint chain.

2. By either optimizing the tangent or the adjoint chain we restrict ourselves to either forward or backward elimination of edges.

3. We do not update the local extended [adjoint] Jacobians as it would actually be necessary. Once we have multiplied two of them we consider the resulting supernode, thus, not permitting a separate treatment of the single vertices which may actually still exist at the scalar level.

The restriction to the computation of an optimal parenthesization of the tangent [adjoint] chain induces a subgraph $M_{\mathbf{T}}$ [$M_{\mathbf{A}}$] of the metagraph M. In our discussion we will concentrate on the optimization of the tangent chain with respect to the number of multiplications required to calculate the chained matrix product. Similar statements hold for the adjoint chain.

Notice that the tangent chain yields decreasing indices when going from the left to the right. This is due to the fact that the forward elimination of an edge (i, j) from the c-graph corresponds to the matrix product $B_j B_i$. Therefore, the indices in both Figures 8 and 7 have to be reversed when regarding edge elimination in c-graphs. Simply set

$$A_1 := B_{p+m}, \ \ldots, \ A_{p+m} := B_1.$$

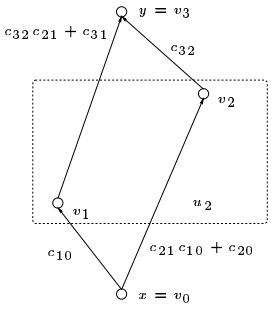

Figure 11: $B_2 B_1$ (2)

We have already noticed that the number of vertices in $M_{\mathbf{T}}$ grows with the number k of factors in the chained matrix product exponentially. However, analogous to the classical chained matrix multiplication problem we can make use of both the optimal substructure principle and the overlapping subproblems. In the course of this it will be necessary to compute the sparsity of the local extended Jacobians that are associated with every vertex in the table look up graph T. We end up with subproblems the number of which is polynomial in k by tabulating the solution of each of these subproblems in T. In Figure 8 we have marked the different subproblems using dashed frames. For example, (1((23)4)) and (1(2(34))) result in one and the same problem as both of them are based on the solution of the subproblem to determine an optimal parenthesization for (234). The latter has already been solved before and its solution can be looked up in T.

Analogous to the classical chained matrix problem let m_{ji} be the minimal number of multiplications required for computing the products $B_j \cdot \ldots \cdot B_i$ for $p + m \geq j \geq i \geq 1$. Then we get

$$m_{ji} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m_{ki} + m_{j(k+1)} + p_{[j(k+1)][ki]}\} & \text{if } j > i \end{cases}$$

where $p_{[j(k+1)][ki]}$ is the cost of multiplying $B_{[j]}$ by $B_{[k]}$. Obviously, as running the chained matrix product represents an edge elimination sequence, the vertices of $M_{\mathbf{T}}$ have unique

equivalents in M. Every edge in $M_{\mathbf{T}}$ can be associated with a set of paths connecting the corresponding vertices in the general metagraph.

We can summarize the above discussion as follows: *Using the tangent chain as a restriction on the metagraph M induces a subgraph $M_{\mathbf{T}}$ the number of vertices in which grows exponentially with the number of factors $(p+m)$ in the chained matrix product. By exploiting the optimal substructure property and tabulating the solutions of overlapping subproblems dynamic programming provides a way to solve the shortest path problem in $M_{\mathbf{T}}$ in a time which is polynomial in $p+m$.*

Example

For the example c-graph shown in Figure 10 we have

$$B_1 = \begin{pmatrix} 1 & & & \\ c_{10} & 0 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}, \quad B_2 = \begin{pmatrix} 1 & & & \\ & 1 & & \\ c_{20} & c_{21} & 0 & \\ & & & 1 \end{pmatrix}$$

and

$$\bar{B}_1 = \begin{pmatrix} 1 & & & \\ & 0 & & \\ & c_{21} & 1 & \\ & c_{31} & & 1 \end{pmatrix}, \quad \bar{B}_2 = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & \\ & & c_{32} & 1 \end{pmatrix}.$$

The computation of the Jacobian J can be expressed as

$$J = \mathbf{e}_3^T \cdot B_3 B_2 B_1 \cdot \mathbf{e}_0 = \mathbf{e}_3^T \cdot \bar{B}_2 \bar{B}_1 \bar{B}_0 \cdot \mathbf{e}_0$$

leading to two chained matrix multiplication problems each of them having two degrees of freedom which can be solved separately using our dynamic programming algorithm. Furthermore, we get

$$B_{[2]} = B_2 \cdot B_1 = \begin{pmatrix} 1 & & & \\ c_{10} & 0 & & \\ c_{21}c_{10} + c_{20} & & 0 & \\ & & & 1 \end{pmatrix}$$

which is equivalent to the forward elimination of $(1,2)$ and where $B_{[1]}$ denotes the local extended Jacobian of the supernode [1] which consists of the two elemental vertices v_1 and v_2 (see Figure 11). Analogous,

$$\bar{B}_{[2]} = \bar{B}_2 \cdot \bar{B}_1 = \begin{pmatrix} 1 & & & \\ & 0 & & \\ & & 0 & \\ & c_{32}c_{21} + c_{31} & c_{32} & 1 \end{pmatrix}$$

which is equivalent to the backward elimination of $(1,2)$ and where $B_{[1]}$ denotes the local extended adjoint Jacobian of [1].

The run-time of the dynamic programming algorithm depends on the number of factors in the chained matrix product quadraticly. Thus, we should think about a way to minimize this number in order to make the algorithm work as efficiently as possible. In general, it is not necessary to solve the problem for the scalar c-graph, i.e. for a chained matrix product where each factor corresponds to the local extended [adjoint] Jacobian of one elemental vertex in the c-graph CG. We should rather look for a minimal *quotient graph* representation of CG where each of its supernodes consists of subsequent, mutually unreachable elemental vertices from CG. We will introduce quotient graphs in the next section.

2.4 Computational Quotient Graphs

Definition 2.4 *For a given c-graph $CG = (V, E)$ we define a* **computational quotient graph** *$CG/\sim = (V, V_\sim, E_\sim)$ to be a vector c-graph with the following properties:*

1. $V_\sim = \{u_i \equiv [i]\,, u_i \subset V : \bigcup_i u_i = V,\ \bigcap_i u_i = \varnothing\}$,
2. $\forall (v_i, v_j) \in u_k :\ v_i \not\prec^* v_j$
3. $([i],[j]) \in E_\sim \quad \Leftrightarrow \quad \exists (v_i \in u_i, v_j \in u_j) :\ (v_i, v_j) \in E.$

Corresponding to our introduction in the previous section we call the subsets $u_i \subset V$ or, equivalently, the sets of indices $[i] \subset \{1-n, \ldots, p+m\}$ supernodes. In particular, the set of minimal vertices X, the maximal vertices Y, and other level sets form supernodes. Any partitioning of V into disjoint supernodes which may be regarded as classes induced by the relation $\sim$ defined by Equation (9) determines a unique directed and acyclic quotient graph. A superedge $([i],[j]) \in E_\sim$ connects two vertices $u_i \in V_\sim$ and $u_j \in V_\sim$ if there is at least one edge in CG connecting an element from u_i with one from u_j. We can associate a local extended Jacobian $B_{[i]}$ with every $u_i \in V_\sim$ such that it contains the local sensitivities of all $v_j \in u_i$ with respect to all elements of predecessors of u_i in $CG/\sim$. Analogous, the local extended adjoint Jacobian $\bar{B}_{[i]}$ contains the local sensitivities of all v_k which are members of successors of u_i in $CG/\sim$ with respect to all $v_j \in u_i$. With other words, the labels on edges leading into a supernode $[i]$ can be combined into a local extended Jacobian

$$C_{[i]} = \sum_{j \sim i} C_j.$$

With Equation (6) we get

$$B_{[i]} = \prod_{j \sim i} B_j = \sum_{j \sim i} C_j + I(M)$$

where $M = \{0, \ldots, n+p+m-1\} \setminus [i]$ and where the order in which the B_j's are multiplied does not matter as the multiplication of any two of them is commutative. Similarly, all labels on edges emanating from a supernode $[i]$ can be united in the corresponding local extended adjoint Jacobian $\bar{B}_{[i]}$.

Let $I_{\mathbf{T}} = \{1, \ldots, p+m\}$ $[I_{\mathbf{A}} = \{1-n, \ldots, p\}]$ and let $\mathcal{S}_{\mathbf{T}}$ $[\mathcal{S}_{\mathbf{A}}]$ denote the family of subsets of $I_{\mathbf{T}}$ $[I_{\mathbf{A}}]$ defined by all classes $[i] \subset I_{\mathbf{T}} \cup I_{\mathbf{A}}$ with respect to $\sim$ under the process of eliminating edges in the c-graph. Then the pair $(I_{\mathbf{T}}, \mathcal{S}_{\mathbf{T}})$ $[(I_{\mathbf{A}}, \mathcal{S}_{\mathbf{A}})]$ defines a topological space with topology $\mathcal{S}_{\mathbf{T}}$ $[\mathcal{S}_{\mathbf{A}}]$ and the supernodes of the quotient graph representing the open sets. We find

$$J = Q_m \left[\prod_{[i] \in \mathcal{S}_{\mathbf{T}}} B_{[i]} \right] P_n^T = Q_m \left[\prod_{[i] \in \mathcal{S}_{\mathbf{A}}} \bar{B}_{[i]} \right] P_n^T.$$

We can now define the elimination of a superedge $([i],[j]) \in E_\sim$ in terms of products of the corresponding local extended [adjoint] Jacobians:

$$\begin{aligned} \text{Forward elimination:} \quad & B_{[j]} := B_{[j]} \cdot B_{[i]} \\ \text{Backward elimination:} \quad & \bar{B}_{[j]} := \bar{B}_{[j]} \cdot \bar{B}_{[i]}. \end{aligned}$$

The elimination of an edge $([i],[j]) \in E_\sim$ leads to the unification of its source and its target resulting in a new vertex $(V_\sim \ni)\ u_i := u_i \cup u_j$. This is exactly what happens during the evaluation of the chained matrix product. By performing a multiplication of two local extended [adjoint] Jacobians we eliminate a superedge in the corresponding computational quotient graph followed by the generation of a new vertex which represents the product of the two factors. Based on the number of multiplications involved in the matrix product we define the forward [backward] Markowitz degree of a superedge $([i],[j])$ analogous to the general vector mode which was introduced in Section 2.2, i.e.

$$\begin{aligned} \text{Forward Markowitz degree of } ([i],[j]): \quad & \mathbf{OPS_F}\{([i],[j])\} = d_j d_i \sum_{k \in P_{[i]}} d_k \\ \text{Backward Markowitz degree of}([i],[j]): \quad & \mathbf{OPS_B}\{([i],[j])\} = d_i d_j \sum_{k \in S_{[j]}} d_k \end{aligned}$$

where $P_{[i]}$ $[S_{[j]}]$ stands for set of supernodes preceding [succeeding] u_i $[u_j]$ in $CG/\sim$. Notice the similarity to what we have described in Section 2.2 where the general vector elimination mode was introduced. In $CG/\sim$ every supernode u_i can be associated with a vector operation

$$\Phi_i : \quad I\!R^{d_i^{\mathtt{pred}}} \supseteq D \to I\!R^{d_i}$$

with $d_i^{\mathtt{pred}}$ and d_i defined as at the beginning of Section 2.2. As in the general vector mode we can associate local Jacobians

$$C_{[j][i]} \in I\!R^{d_i \times d_i^{\mathtt{pred}}}$$

with the superedges $([i],[j])$ in $CG/\sim$. As above the forward [backward] elimination of $([i],[j])$ involves $\mathbf{OPS_F}\{([i],[j])\}$ $[\mathbf{OPS_B}\{([i],[j])\}]$ multiplications.

Minimal Quotient Graphs

Both run-time and memory requirements of a dynamic programming algorithm for optimizing the chained matrix product depend on the number of factors in the chain. Therefore, our objective should be to make this number as small as possible by minimizing the number of supernodes in the quotient graph.

Definition 2.5 *A quotient graph $CG/\sim$ of a c-graph CG is said to be* **minimal** *if it contains the minimum number of supernodes.*

Definition 2.6 *The transitive closure of a* **sub-minimal quotient graph** *$CG/\sim$ contains an edge $([i],[j])$ or $([j],[i])$ for any two supernodes $u_i \in V_\sim$ and $w_j \in V_\sim$.*

Obviously, any minimal quotient graph has to be sub-minimal as otherwise it would be possible to combine the two mutually unreachable vertices, thus, getting a smaller graph.

Why do we distinguish between minimal and sub-minimal quotient graphs? Remember that the quotient graph approach represents a restriction on the metagraph, which should help us to reduce the size of the search space of the shortest path problem. Thinking of a general vertex elimination procedure on a given quotient graph it would certainly be advantageous to consider only those quotient graphs where none of the intermediate vertices are mutually independent. Thus, we will avoid any obsolete operation with respect to the shortest path problem as any trivial commutativity has been fully removed. Hence, the definition of sub-minimal quotient graphs is useful as it provides alternatives to the minimal quotient graphs. This could be advantageous for examining the behavior of our dynamic programming algorithm with respect to different subgraphs of the metagraph.

Theorem 2.1 *Let l be the length of a longest vertex path in CG. Then a minimal quotient graph has exactly l vertices.*

Proof:

If a longest vertex path in CG has length l then it is impossible to extract a quotient graph which has less than l supernodes since otherwise the independence criterion for the elements within the supernodes could not be satisfied.

We will show that it is always possible to construct a minimal quotient graph $CG/\sim$ of CG which has exactly l supernodes. Say $(v_{i_1},\ldots,v_{i_l})$ are the vertices of a longest path in CG. Suppose that there is a vertex v_k which cannot be assigned to any of the supernodes defined by the classes $[i_1],\ldots,[i_l]$. Then there would be a semi-path connecting v_k with each vertex in the longest path. However, this would give us a path of length $l+1$ which represents a contradiction to the assumption that $(v_{i_1},\ldots,v_{i_l})$ is a longest path. ◀

Definition 2.7 *Let $\mathcal{I} : V(CG) \to I\!N$ be a consistent numbering of the vertices of a c-graph CG. We call a quotient graph $CG/\sim$ of CG* **$\mathcal{I}$−based** *if all its supernodes consist of subsequent indices with respect to $\mathcal{I}$, exclusively.*

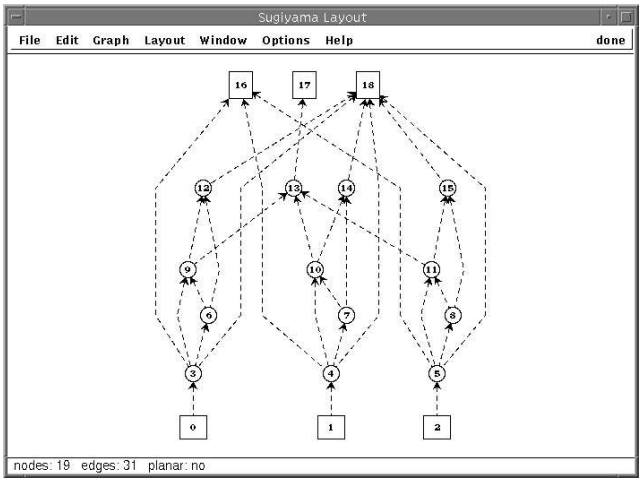

Figure 12: Chebyshev Quadrature, $n = m = 3$, BFF

Notice that the implicit order within the tangent [adjoint] chain depends on the given numbering of the vertices in the c-graph. Hence, the indexing scheme will have a strong impact on the result obtained by solving the chained matrix optimization problem.

Theorem 2.2 *For a given consistent numbering $\mathcal{I} : V(CG) \to I\!N$ of the vertices of CG the solutions to the chained matrix problems with respect to the scalar graph and any $\mathcal{I}-$based quotient graph are equivalent.*

Proof:

If $B_{[j]}B_{[i]}$ is supposed to be optimal then by the *optimal substructure* criterion of dynamic programming both the parenthesizations for $B_{[j]}$ and $B_{[i]}$ have to be optimal, too. Representing supernodes from $V_\sim$ they certainly are as their elements are mutually independent. ◀

The result of running the dynamic programming algorithm on a c-graph CG depends on the numbering of the vertices of CG. Every numbering $\mathcal{I}$ uniquely defines a scalar tangent [adjoint] chained matrix product. If we allow *general* quotient graphs which are not necessarily $\mathcal{I}-$based then we may get worse results by running the dynamic programming algorithm on them compared to the scalar version. We have implemented two indexing schemes which are both consistent and deliver sub-minimal quotient graphs in any case. The *Breadth First Forward* (BFF) numbering applied to the Chebyshev Quadrature problem from the `MINPACK` test problem suite [ACM91] is illustrated in Figure 12. Analogous, a *Breadth First Reverse* (BFR) strategy also delivers a well-defined numbering of all vertices in CG which corresponds to Definition 2.3. In general, the numberings resulting from the application of BFF and BFR are not the same.

3 Case Study

With the help of the Chebyshev Quadrature problem we will explain the features which our software named **OESCOMP** offers for the optimization of the chained matrix products. For further information on the program refer to [Nau99a] Currently, the user can choose between the following four versions of dynamic programming algorithms:

BFFT: Tangent chain on BFF-based $CG/\sim$,
BFFA: Adjoint chain on BFF-based $CG/\sim$,
BFRT: Tangent chain on BFR-based $CG/\sim$,
BFRA: Adjoint chain on BFR-based $CG/\sim$.

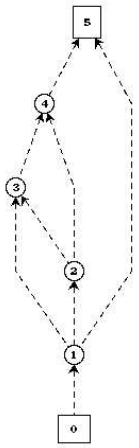

Figure 13: $CG/\sim$

Apart from the c-graph which is shown in Figure 12 of the previous section it is also possible to visualize the computational quotient graph and the table look up graph that the dynamic programming algorithm is built on. It is easy to check that the extraction of $CG/\sim$ for the Chebyshev Quadrature problem with $n = m = 3$ leads to a chained matrix multiplication problem with five factors. The corresponding BFF-based quotient graph is shown in Figure 13.

Figure 14 shows the table look up graph T for the above problem. The `C++` library `LEDA` [MeNä96] provides the possibility to assign user defined data to the nodes representing the different stages of the dynamic programming algorithm. In **OESCOMP** the user can select nodes of T interactively and visualize their characteristics as shown in Figure 14 for node number 10. The `data` field in the `Node Setup` window contains the following information:

- Node 10 is a member of the second level of T.
- At the stage in the metagraph which is associated with node 10 all edges of $CG/\sim$ between supernodes containing vertices with indices between 6 and 15 in CG have been eliminated.
- It took 12 multiplications to perform this part of the chained matrix product as efficiently as possible.
- The first out of two possible parenthesizations of the internal subproblems gave this result. This information is used for the construction of the optimal full parenthesization.

The following table shows the entire optimization process for the **BFFT** method. There are five levels (0-4) in the table look up graph with the number of nodes decreasing by one from level to level. With the help of the information provided it is straight forward to recapitulate

the way our dynamic programming algorithm works.

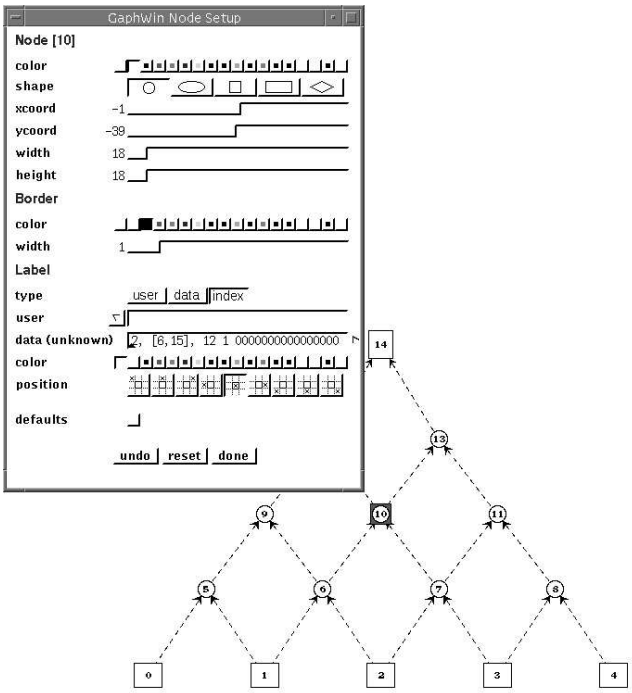

Figure 14: Table Look Up Graph

0	1	2	3	4
$[3,5],0$				
$[6,8],0$	$[3,8],3$			
$[9,11],0$	$[6,11],3$	$[3,11],9$		
$[12,15],0$	$[9,15],12$	$[6,15],12$	$[3,15],18$	
$[16,18],0$	$[12,18],9$	$[9,18],21$	$[6,18],18$	$[3,18],27$

Finally, the algorithm delivers the minimal number of multiplications required for running the chained matrix product this way (27 for the above example). The corresponding optimal parenthesization is extracted recursively by applying the optimal substructure principle backwards.

4 Numerical Results

We have applied the proposed method to more than a hundred test problems out of which we will present a small but representative subset. In the previous section we have already looked at one particular test problem. Now we will list a few results without presenting a detailed discussion for each of them. The numbers will speak for themselves. We will consider the following eleven test problems:

PDJ:	Pressure distribution in a journal bearing problem;
SSC:	Steady state combustion problem;
FDC:	Flow in a driven cavity problem;
FCH:	Flow in a channel problem;
WAT:	Watson function;
DIE:	Discrete integral equation function;
VDI:	Variably dimensioned function;
PEN:	Penalty function II;
EXP:	Extended Powell function;
SPE:	Speelpenning function;
GDF:	Gaussian data fitting problem.

Apart from the Speelpenning function [Spe80] the above examples are taken from the MINPACK test problem suite [ACM91]. In the following table we will compare the values delivered by the dynamic programming optimization method **BFRA** with the theoretical operations counts that can be achieved using state-of-the-art AD-technology (see Section 1). $[\hat{n}, \hat{m}]$ denotes the minimum out of the maximal numbers of non-zero elements per row and per column of the Jacobian.

	n	p	m	$[\hat{n}, \hat{m}]$	**DF**	**DR**	**NR**	**BFRA**	**NR / BFRA**
PDJ	64	1447	1	1	92672	1511	1511	1278	**1.18**
SSC	16	655	1	1	10496	671	671	452	**1.49**
FDC	16	984	16	11	16000	16000	11000	930	**11.83**
FCH	32	1209	32	9	39712	39712	11169	845	**13.22**
WAT	7	1683	7	7	11830	11830	11830	4240	**2.79**
DIE	20	2499	20	20	50380	50380	50380	1659	**30.34**
VDI	100	504	100	100	60400	60400	60400	10301	**5.86**
PEN	200	2399	1	1	480000	2599	2599	1798	**1.45**
EXP	96	479	1	1	4680	575	575	528	**1.09**
SPE	50	48	1	1	2450	98	98	96	**1.02**
GDF	11	1625	65	11	18590	106340	18590	1430	**13.0**

Considering the ratio between the optimal one-sided Newsam-Ramsdell approach and the **BFRA** dynamic programming method we observe that savings within the range from nearly 3 up to 30 can be achieved. Obviously, this cannot be the case for the computation of single gradients. However, for $\{n, m\} \gg 1$ we get a significant decrease of the number of multiplications involved in the accumulation of the Jacobian for virtually all problems.

For most real-world problems the generation of optimized derivative code based on either forward or backward vertex elimination sequences combined with the pre-elimination of all hoisting vertices would result in remarkable savings in the overall operations count. The fact that it is not clear a priori whether we should prefer the forward or the backward approach is one of the problems. At this point it could be useful to exploit the strength of heuristics for determining nearly optimal vertex elimination sequences for almost all sorts of c-graphs. **V_LR** turned out to be very consistent, while being only slightly more expensive than the pure uni-directional methods. It could be worth to analyze selected evaluation routines deeper by using more costly optimization methods like dynamic programming or simulated annealing. However, the additional effort is justified only if the resulting derivative code is generated once and is then used over and over again.

5 Summary, Conclusion and Outlook

The goal of this paper was to present an approach to the efficient accumulation of Jacobian matrices using elements from graph theory and from combinatorial optimization. The application of the chain rule to c-graphs is interpreted as the elimination of edges. By successively eliminating all intermediate edges we get to a stage where the c-graph represents a subgraph of a complete bipartite graph $K_{n,m}$, with a bipartition that corresponds to the n independent and the m dependent variables. The labels on the remaining edges are then exactly the non-zero entries of the complete Jacobian. Depending on whether the chain rule is applied as usual or to the adjoints, one distinguishes between two equal ways of eliminating edges which are referred to as forward and backward. The number of multiplications involved in the forward [backward] elimination of an edge is called its forward [backward] Markowitz degree. The sum of the forward [backward] Markowitz degrees of all edges at the time of their elimination from the c-graph (overall Markowitz degree) is considered to be the cost of calculating the complete Jacobian. In order to minimize this cost we have to solve a shortest path problem on the metagraph the vertices (stages) of which stand for all different graphs that can be constructed starting with the original c-graph by applying an arbitrary sequence of both forward and backward edge eliminations. Since the number of stages in the metagraph grows exponentially with the number of intermediate vertices in the original c-graph we have to put certain restrictions on the metagraph in order to reduce the size of the search space of the shortest path problem. This approach leads to different subgraphs of the metagraph which are then subject to the same shortest path problem, while having the number of distinct paths to be checked reduced.

The restriction to the elimination of vertices is one practicable way to reduce the size of the metagraph. However, the number of stages in the resulting subgraph still grows exponentially with the number of intermediate vertices in the original c-graph. We conjecture that the vertex-edge discrepancy is less or equal to 2. Therefore, we have focussed on vertex elimination for the development of heuristics for solving the shortest path problem in the metagraph.

We have presented dynamic programming algorithms for optimizing the chained local extended [adjoint] Jacobians product the complexity of which is polynomial in the number of intermediate vertices in the c-graph. Most of our theoretical results were implemented and tested on numerous example problems.

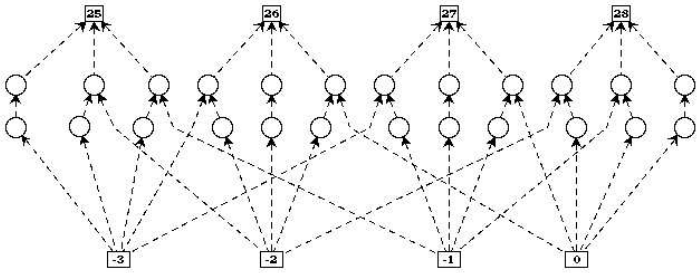

Figure 15: Solid Fuel Ignition Problem

The exploitation of the results of the cross-country elimination method should lead to the automatic generation of optimized derivative code. Since memory accesses have a strong impact on the run-time of the code we should concentrate on locally bounded parts of the c-graph in order to make full use of the reduction in the operations count. Hierarchical vertex elimination could be one way to ensure this.
We are not able to generate optimized derivative code automatically, so far. Therefore, we have decided to compare the run-times of a **BFRA**-based hand-written code (`Code 2`) with a scalar tangent code without exploitation of sparsity (`Code 1`) for the Solid Fuel Ignition problem [ACM91]. Figure 15 shows the corresponding c-graph. Regarding the number of multiplications involved in the computation of the (4×4)-Jacobian it is possible to achieve savings by a factor of 4:

n	p	m	$[\hat{n}, \hat{m}]$	**DM**	**NR**	**V_LR**	**DM / V_LR**
4	24	4	3	128	84	32	**4**

Using the corresponding derivative codes presented in [Nau99a] we will check whether these savings can be transfered to decreases in the run-time by a comparably high factor. We have considered the elapsed CPU times for 10^5 successive calculations of the complete Jacobian on a SUN Sparc20:

	`Code 1`	`Code 2`	`Code 3`	`Code 4`	`Code 1 / Code 2`
t_{CPU} (in sec)	9.8	2.8	2.6	3.15	**3.5**

In addition to codes 1 and 2 we have included the optimal hand-coded Jacobian (`Code 3`) and the derivative code supplied by the `MINPACK` package. Obviously, for this small problem the memory accesses will not have any negative impact on the run-time of cross-country elimination sequences. This results in the factor of 3.5 by which `Code 2` ran faster than `Code 1`. It remains to mention that while being a very robust method for computing nearly optimal elimination sequences the dynamic programming algorithm is rather expensive in terms of run time compared to the simple forward and reverse modes. However, if the generated optimized derivative code is used many times this additional effort could pay off soon.

To summarize the above, it appears to be useful to work on the automatic generation of optimized adjoint code based on different elimination sequences. In order to be able to handle large-scale evaluation programs the hierarchical approach has to be implemented efficiently. Therefore, we require tools for analyzing the code which generate some (ideally standardized) intermediate form which contains all the necessary information [Bro98], [BRM96].

From the theoretical point of view, the proof of the NP-completeness of the general edge elimination problem is still not given. Our conjecture about the small constant vertex-edge discrepancy has a more practical relevance. To support the search for its proof the implementation of a simulated annealing algorithm for optimizing edge elimination sequences could be useful. The principle is the same as for vertex elimination sequences. We simply have to adapt the annealing schedule such that rearrangements including both forward and backward elimination of edges become possible.

The current version of our software package **OESCOMP** is to be regarded as experimental. Since it represents a useful tool for supporting both research and teaching in the field of AD we would like it to be subject to further development.

We believe that the efficient implementation of different edge elimination algorithms that lead to the automatic generation of optimized derivative code will take AD a large step forward. We expect this to influence both the degree of fame as well as the acceptance of AD positively.

References

[ACM91] B. Averik, R. Carter, and J. More, *The Minpack-2 test problem collection (preliminary version)*, Technical Memorandum No. 150, Mathematical and Computer Science Division, Argonne National Laboratory, 1991.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[BBCG96] M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds, *Computational differentiation: techniques, applications, and tools*, SIAM, Philadelphia, PA, 1996.

[Bis96] C. H. Bischof, *Hierarchical approaches to automatic differentiation*, in [BBCG96], pp. 83–94.

[BRM96] C. Bischof, L. Roh, and A. Mauer, *ADIC: An extensible automatic differentiation tool for ANSI-C*, Preprint ANL/MCS-P626-1196, Argonne National Laboratory, March 1997.

[Bro98] S. Brown, *Models for automatic differentiation: A conceptual framework for exploiting program transformation*, PhD thesis, Computer Science, University of Hertfordshire, Hatfield, England, February 1998.

[CoGr91] G. Corliss and A. Griewank, eds, *Automatic differentiation: theory, implementation, and application*, SIAM, Philadelphia, PA, 1991.

[GrRe91] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in [CoGr91], pp. 126-135.

[MeNä96] K. Mehlhorn and S. Näher, *LEDA, a platform for combinatorial and geometric computing*, Communications of ACM, Vol. 38, no. 1, pp. 96-102, 1995.

[Nau99a] U. Naumann, *Efficient calculation of Jacobian matrices by optimized application of the chain rule to computational graphs* Ph.D. thesis, Institute for Scientific Computing, Dresden University of Technology, 1999.

[Nau99b] U. Naumann, *Optimizing the Accumulation of Jacobians by Edge Elimination in Computational Graphs* INRIA Rapport de Recherche Nr. 3659, Institut National de Recherche en Informatique et Automatique, Sophia-Antipolis, France, 1999.

[Nau99c] U. Naumann, *SAVE - Simulated Annealing applied to the Vertex Elimination Problem in Computational Graphs* INRIA Rapport de Recherche Nr. 3660, Institut National de Recherche en Informatique et Automatique, Sophia-Antipolis, France, 1999.

[NeRa83] G. Newsam and J. Ramsdell, *Estimation of sparse Jacobian matrices*, SIAM J. Alg. Discr. Meth., 4 (1983), pp. 404-417.

[Spe80] B. SPEELPENNING, *Compiling fast partial derivatives of functions given by algorithms*, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, January 1980.

[Wen64] R. E. WENGERT, *A simple automatic derivative evaluation program*, Comm. ACM, 7 (1964), pp. 463-464.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399