Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt
der Fakultät Mathematik und Naturwissenschaften der
Technischen Universität Dresden

von

Dipl.-Math. Uwe Naumann

geboren am 24. Juni 1969 in Rodewisch

Gutachter:	Prof. A. Griewank
	Prof. B. Christianson
	Prof. K. Kubota

Eingereicht am:	27. November 1998
Tag der Verteidigung:	12. Februar 1999

To My Parents

June 18, 1999

Above all I would like to thank Prof. Andreas Griewank for his helpful support during my research. His inspiration and his advice formed the foundation for this thesis. The informal and friendly atmosphere at the Institute of Scientific Computing created a very pleasant daily working environment. My gratitude is for all my colleagues at the Institute for their interest and useful discussions. Especially, I want to thank Andrea Walter, Hristo Mitev, and Olaf Vogel for helping me to put this thesis into shape.

Contents

Preface

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 $F : I\!R^n \supseteq D \to I\!R^m$ given in form of a computer program numerically up to machine precision. 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 results in 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. Furthermore, there is no generally accepted terminology for dealing with these combinations of the forward and reverse modes of AD, so far.

Our objective is to compute J as fast as possible. Therefore, we will focus on the minimization of the number of scalar multiplications involved in this process, by finding optimal application sequences of the chain rule. The here presented approach must be seen in the context of making AD more efficient and it is supposed to provide the users a deeper insight into the structure of problems. Our goal is to provide the framework for the search of a solution of the computationally hard combinatorial optimization problem which is to minimize the number of scalar multiplications involved in the accumulation of the complete Jacobian. Furthermore, we intent to develop several alternative methods for approximately solving the above problem which, apart from their theoretical relevance, could also be implemented into existing or future software packages for AD.

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]. So far, there are nearly no results on the potential reductions of the operations counts achievable by different methods for determining elimination sequences. The work by Griewank and Reese [GrRe91] examines the so-called *Markowitz criterion*, which has its origins in the Gaussian elimination of sparse matrices, more closely. It also presents numerical results on the savings in the number of multiplications for accumulating the Jacobian matrix. Tests show that the Markowitz heuristic fails on a large number of problems. Thus, it is certainly useful to develop new, more robust techniques for efficiently computing Jacobians.

This thesis will start with a brief introduction into Automatic Differentiation focusing on its representation in the form of computational graphs. The *chain rule* will be interpreted as the elimination of intermediate edges. We will show that, in general, edge elimination is superior (with respect to the number of multiplications performed when computing the Jacobian) to vertex elimination which has been considered to be elemental until now. The concept of *linearized computational graphs* will be extended to vector valued elemental functions thus allowing a hierarchical application of the methods to be developed. The combinatorial optimization problem of determining an edge elimination sequence that minimizes the number of multiplications required to accumulate the complete Jacobian of the given vector function will be formulated as a shortest path problem in the *metagraph*. We will prove two theoretical results concerning bounds for the numbers of edges leading into and emanating from the intermediate vertices at any stage in the metagraph.

Chapters 3, 4 and 5 represent the main part of this thesis as they describe different new or adapted methods for solving the combinatorial optimization problem heuristically. These methods contain local heuristics as well as dynamic programming algorithms and a collection of simulated annealing schedules for the computation of nearly optimal vertex elimination sequences. All the approaches proposed here are implemented as a `C++` program named **OESCOMP**. Numerous test results will support our observations. With the help of an example we will show how the reduction of the operations count can be transferred into optimized derivative code which finally results in the speed up that we are actually aiming for.

Chapter 1

The Chain Rule on Graphs

1.1 From Programs to Graphs

The computation of the Jacobian matrix of a vector function

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

is essential in many numerical algorithms. In our approach we expect F to be such that it can be decomposed into a sequence of scalar valued functions

$$\varphi: \quad \mathbb{R}^d \supseteq D_\varphi \to \mathbb{R}$$

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**. Obviously, the basic arithmetic operations $\{+, -, *, /\}$ and the elementary functions $\{\sin, \cos, \exp, \log, ...\}$ which are provided by most high-level programming languages are elemental. In fact, we observe that most intrinsics represent a special subset of the elemental functions, having the length d of their argument vectors bounded by two.

1.1.1 Evaluation routines

In most cases the vector function F is given as a computer program written in some high-level programming language such as `C`, `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

$$v_j = \varphi_j(v_i)_{i \in P_j} \quad \in \mathbb{R}, \tag{1.2}$$

by assigning the result of every elemental function φ_j that occurs in the program to a unique intermediate variable v_j. By P_j [S_j] we denote the set of indices of all predecessors [successors] of the intermediate variable v_j with respect to the implicit order within the evaluation routine. Thus, φ_j acts on an argument vector $\mathbf{u} \in \mathbb{R}^d$ where $d = |P_j|$. The fact that the result v_j of an elemental function φ_j depends on some argument v_i directly is denoted by $v_i \prec v_j$ and the transitive closure of this relation by $v_i \prec^* v_j$.

Assumption 1.1.1 *We assume the function of interest to be fixed in the sense that there are no conditional branches in it.*

By unrolling loops and by substituting calls to subroutines with the corresponding code it is possible to assign the result of every elemental function to a unique intermediate variable. According to Equation (1.2) we can represent this evaluation routine as the corresponding sequence of scalar assignments.

Definition 1.1.1 *An evaluation routine of a vector function as in Equation (1.1) is in form of a so-called* **evaluation trace** *(also* **single assignment code***) if it is written as the sequence of scalar assignments to the corresponding intermediate variables*

$$v_{1-n} \equiv x_0, \dots, v_0 \equiv x_{n-1}, v_1, \dots, v_q, v_{q+1} \equiv y_0, \dots, v_{q+m} \equiv y_{m-1}$$

given by Equation (1.2).

Reformulating Assumption 1.1.1, we expect the evaluation trace of a given vector function F not to depend on the argument at which the value of F is calculated. For a given evaluation trace we distinguish between the n independent, $X = \{v_{1-n}, \dots, v_0\}$, the m dependent, $Y = \{v_{q+1}, \dots, v_{q+m}\}$, and q intermediate variables $Z = \{v_1, \dots, v_q\}$, where Z and Y represent the results of the $q+m$ elemental functions in F.

Assumption 1.1.2 *The sets X, Z, and Y are disjoint.*

Keeping in mind, that our objective is to calculate the complete Jacobian of a given vector function which consists of the partial derivatives of the m dependent variables $y_0, \dots, y_{m-1}$ with respect to each of the n independent variables $x_0, \dots, x_{n-1}$ it is fundamental to assume the following:

Assumption 1.1.3 *Given an evaluation routine of a vector function defined by Equation (1.1), for some fixed argument the elemental functions φ_j are well defined for $j = 1, 2, \dots, q+m$ and have jointly continuous partial derivatives*

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

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

1.1.2 Computational graphs

The relation between the variables in a given evaluation routine can be visualized by a graph. Since we will deal exclusively with graphs representing a certain underlying vector function, which are always directed and acyclic, we are going to define the term graph in a correspondingly suitable way. This prevents us from the necessity of repeatedly stating the whole list of attributes which are the same in any case. Below, we will introduce the complete terminology which we will build on when exploring graphs associated with evaluation routines of vector functions.

Terminology

A (**finite, weakly connected, directed, acyclic**) **graph** (dag) $G = (V, E)$ consists of a finite set V of **vertices** (nodes) v_i and a collection E of ordered pairs of distinct vertices (v_j, v_k), called the **edges** (arcs) of G, which we will write as the corresponding ordered pairs of indices (j, k). To be able to explain the remaining features, connected and acyclic, we will need some further framework which is given below.

A **walk** in a directed graph is an alternating sequence of vertices and edges, $v_0, (0, 1), v_1, \ldots, v_{n-1}, (n-1, n), v_n$, with edges $(i-1, i)$. A **path** is a walk in which all vertices are distinct. A **cycle** is a non-trivial closed walk with equal first and last vertices and all remaining vertices being distinct. Analogous to a walk a **semi-walk** is again an alternating sequence of vertices and edges, but now each edge may be either $(i-1, i)$ or $(i, i-1)$. In the same way we define semi-paths, semi-cycles, and so forth.

Now we can complete the definition of the term graph as we are going to use it. A graph is also (weakly) connected, which means that every two points are joined by a semi-path, and it is acyclic, implying that G does not contain any cycle.

If (i, j) is an edge in G then we say that it is **incident with** v_i and v_j. We call v_i the **source** of the edge (i, j) and v_j its **target**. In this case v_i is **adjacent to** v_j and v_j is **adjacent from** v_i. (i, j) is said to be an **in-edge** of v_j and an **out-edge** of v_i. The **converse digraph** G' of G has its edges reversed.

For every vertex $v_i \in G$ we define its **in-degree** as the number of edges having v_i as its target and we call the number of edges with source v_i the **out-degree** of v_i. If we denote the set of source-[target-]indices of edges leading into [emanating from] v_i by P_i $[S_i]$ it is useful to introduce the notation $|P_i|$ $[|S_i|]$ for the in-degree [out-degree] of v_i. We call a vertex $v_i \in G$ having in-degree [out-degree] zero **source** or **minimal** [**sink** or **maximal**]. If there is a path from v_i to v_j then v_j is said to be **reachable from** v_i.

A (directed) **tree** is a graph without semi-cycles. A **subgraph** of G is a graph having all of its vertices and edges in G. A subgraph of G with just one source [sink] is called **out-tree** or **arborescence [in-tree]**. If a subgraph of G contains all vertices of G it is called **spanning subgraph**. Let $R_i \subseteq V$ be the set of all vertices in G that are reachable from a vertex v_i. Then a **spanning** v_i**-arborescence** is an out-tree with source v_i which contains all vertices of R_i.

In order to optimize the vocabulary it is useful to give two more different definitions of the term path. A sequence $(i_0, \ldots, i_l) \equiv ((i_0, i_1), \ldots, (i_{l-1}, i_l))$ of edges is called an **edge path** of length l connecting its **source** v_{i_0} with its **target** v_{i_l} if the target of every edge is equal to the source of its successor in $(i_0, \ldots, i_l)$. On the other hand, a sequence of vertices $(i_0, \ldots, i_l) \equiv (v_{i_0}, v_{i_1}, \ldots, v_{i_l})$ is called a **vertex path** of length $l+1$ connecting v_{i_0} with v_{i_l} if any two successive vertices $v_j, v_k \in (i_0, \ldots, i_l)$ are connected by an edge (j, k). For each edge path there is an equivalent vertex path. A particular **path** $(i_0, \ldots, i_l)$ can be represented either as an edge path or as the corresponding vertex path. Two paths are said to be **edge-disjoint [vertex-disjoint]** if they do not have any edge [vertex] in common.

A set $D \subseteq E$ of edges in G with the property that any path from a vertex v_i to a vertex v_j includes an edge of D is called $[v_i|v_j]$**-disconnecting set**. $S \subseteq V \backslash \{v_i, v_j\}$

is a $[v_i|v_j]$-**separating set** of G if any path from v_i to v_j passes through a vertex of S. The maximum number of vertex-disjoint paths connecting two distinct, non-adjacent vertices v_i and v_j in a graph G is called **vertex-connectivity index** of v_i and v_j and is denoted by $c(v_i, v_j)$. The minimum number of vertices in a $[v_i|v_j]$-separating set in a graph G is called **vertex-separability index** of v_i and v_j and is denoted by $s(v_i, v_j)$. We will use two kinds of matrices for exploring graphs:

1. **Adjacency matrix** $A(G) = (a_{ji})$ with

$$a_{ji} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} .$$

   $A(G)$ is a lower triangular $(q \times q)$-matrix for a graph containing q vertices.

2. **Reachability matrix** $R(G) = (r_{ji})$. For a dag G $R(G)$ is also a lower triangular $(q \times q)$-matrix with

$$r_{ji} = \begin{cases} 1 & \text{if } (i, \ldots, j) \in G \\ 0 & \text{otherwise} \end{cases} .$$

   $R(G)$ contains a nonzero entry $r_{ji} = 1$ for all vertices v_i connected with v_j by a path in G.

It is advantageous to distinguish between different forms of graphs representing an evaluation routine of a vector function F. From the practical point of view the user of AD software will always have "some code" that he wants to differentiate "in some way". He cannot be sure that the *evaluation code* is "clean" in the sense that it contains only statements that are needed for the computation of F. Knowing the independent and the dependent variables a dependency analysis will be required in order to extract what we will finally refer to as the *computational graph*. So, the hierarchy that will be introduced with respect to the computational graphs is not crucial from the pure mathematical point of view. However, it makes the steps that have to be taken in order to implement our results easier to understand.

Given an evaluation code that is supposed to define a vector function F as in Equation (1.1) we distinguish between five types of vertices:

- X: set of minimal vertices representing the independent variables,
- Y: set of maximal vertices representing the dependent variables,
- Z: set of vertices representing the intermediate variables,
- D_1: set of vertices representing variables which do not depend on any of the independent variables,
- D_2: set of vertices representing variables which belong to computational dead-ends, i.e. there is no dependent variable depending on any variable represented by a vertex in D_2.

Notice that with respect to the notation there is no need to distinguish between the vertices in the computational graph and the variables in the corresponding evaluation routine.

Obviously, the variables belonging to computational dead-ends exclusively are not part of the definition of F and may thus be omitted. The **complete computational graph**

$$CG_F^c = (V^c, E^c), \quad \text{where} \quad V^c = X \cup Z \cup Y \cup D_1,$$

is a graph in the sense of our definition representing a certain underlying evaluation routine of a given vector function F. It is acyclic since the evaluation of F could not be well defined if two intermediate variables v_i and v_j were mutually dependent on each other. We may consider CG_F^c as representing the linearization of the composite function given by Equation (1.1) at the current argument. CG_F^c is used for the computation of the values of the intermediate and dependent variables at the current argument and for calculating the local partial derivatives c_{ji} labeling the edges $(i,j) \in E^c$. Both computations can be performed during one so-called **initial forward sweep** through the complete computational graph CG_F^c. This gives us all local partial derivatives that are required for computing the entries of the complete Jacobian by exploiting the chain rule.

Our objective is to minimize the number of scalar multiplications required to compute the complete Jacobian of F. As will be described in Section 1.2 it is therefore necessary to consider all edges lying on paths which connect the minimal vertices, representing the independent variables, with the maximal vertices, standing for the dependent variables of F in CG_F^c. Hence, following the initial forward sweep we may disregard all vertices belonging to D_1 when working on a solution of our problem. Thus, we get to the (reduced) computational graph

$$CG_F = (V, E), \quad \text{where} \quad V = V^c \setminus (D_1 \cup D_2),$$

containing all information needed to be able to compute the entries of the Jacobian. From now on we will refer to CG_F as the **computational graph** (also **c-graph**) and whenever it is clear that we deal with a certain function F, we skip the subscript and denote the c-graph simply by CG. We will distinguish between $n \equiv |X|$ minimal [independent], $p \equiv |Z|$ intermediate and $m \equiv |Y|$ maximal [dependent] vertices and analogous to the evaluation trace we will enumerate them as follows:

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

Definition 1.1.2 *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.*

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

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. According to Assumption 1.1.3 labels c_{ji}, representing the local partial derivatives, are attached to all edges $(i,j) \in E$. The $v_i \in X$ are the only minimal and the $v_j \in Y$ the only maximal vertices in CG. All intermediate vertices $v_k \in Z$ lie on a path connecting an independent with a dependent vertex. We observe that v_i depends on v_j in F if and only if there exists a path connecting v_i with v_j in CG.

1.2 From Graphs to Jacobians

So far, we have discussed some basic principles of evaluation routines and their representation as evaluation traces and as c-graphs. Our objective is to compute the complete Jacobian J of a vector function F given by Equation (1.1). Based on F's graph representation we require a method of transforming the c-graph in a way such that we get something equivalent to J. 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

$$c_{ji} \equiv \frac{\partial}{\partial x_i} y_j \qquad \text{for } i = 0, \ldots, n-1 \text{ and } j = 0, \ldots, m-1$$

on the *final edges* (see Section 1.2.2), connecting the minimal vertices with the maximal ones, are exactly the nonzero entries of the Jacobian matrix of F. This situation is shown in Figure 1.9. But what does *elimination of an intermediate vertex [edge] from the c-graph* mean? The answer to this question will be given in this section. We will start with a brief introduction into the basics of AD based on two different interpretations of the chain rule. The ideas will be applied to c-graphs, resulting in rules for eliminating both edges and vertices.

1.2.1 Fundamentals of AD

AD is based on the application of the **chain rule** which is illustrated in Figure 1.1. Considering the dependency of v_j on the independent variables x_i we can calculate $\partial v_j/\partial x_i$ as the sum over v_j's predecessor's sensitivities $\partial v_l/\partial x_i$, $l \in P_j$, each of them multiplied by the corresponding local partial derivative c_{jl} of the elemental function φ_j with respect to the variables v_l. Analogous, we can look at the dependency of y_k on the assumed independent variable v_j. In this case we compute $\partial y_k/\partial v_j$ as the sum over y_k's sensitivities with respect to the successors of v_j, again multiplied with the local partial derivatives of the elemental functions φ_l, $l \in S_j$. The exploitation of these two different interpretations of the chain rule will lead to the forward and reverse modes of AD as well as it serves as the basis for the more general cross-country elimination approach.

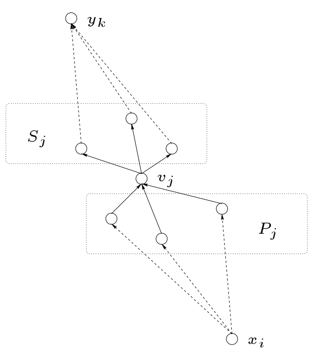

Figure 1.1: Chain Rule

Building on the representation of the vector function F as an evaluation trace in connection with Assumption 1.1.3 about the differentiability of the occurring elemental functions follows by the chain rule that F is well defined and once continuously differentiable. Its Jacobian matrix $J(\mathbf{x}) \equiv \nabla_{\mathbf{x}} F(\mathbf{x}) \in I\!R^{m \times n}$ is computable from the local partial derivatives c_{ji}.

Brief descriptions of the forward and reverse modes of AD are given below. To begin with, we will concentrate on the simplest case, the so-called *scalar mode on* *the scalar c-graph*. A general classification of the different modes running on various types of c-graphs is given in Section 1.3.

Our short introduction will not be exhaustive by far. Those who are not familiar with the subject should read the introduction into AD by Rall and Corliss [RaCo96] published in the proceedings of the Second International Workshop on Computational Differentiation held in Santa Fe, USA, in February 1996 or they should refer to the paper by Iri [Iri91] on *History of Automatic Differentiation and Rounding Error Estimation* in the proceedings of the first workshop on AD held in Breckenridge in 1991. More advanced aspects of AD are covered by [Chri93], [Kub91], [GUW97], [BKBC96], and others.

Forward mode

The (scalar) **forward mode** of AD is based on the application of the chain rule in the usual way. Consider an evaluation routine given in form of a single assignment code and let again P_j denote the set of all indices $l < j$ such that v_j depends directly on $v_l \in X \cup Z$. Then the chain rule takes the form

$$\frac{\partial v_j}{\partial x_i} = \sum_{l \in P_j} c_{jl} \cdot \frac{\partial v_l}{\partial x_i} \qquad \text{for} \quad j = 1, \dots, p+m \quad \text{and} \quad i = 0, \dots, n-1$$

to compute the partial derivatives of the intermediate and dependent variables with respect to each of the independent variables. The substitution $\dot{v}_j \equiv \partial v_j / \partial x_i$ leads to the following equation which describes the forward propagation of tangents exploiting the chain rule:

$$\dot{v}_j = \sum_{l \in P_j} c_{jl} \cdot \dot{v}_l \qquad \text{for} \quad j = 1, \dots, p+m. \tag{1.3}$$

The initialization of $\dot{\mathbf{x}}$ followed by one forward sweep corresponding to Equation (1.3) gives us $\dot{\mathbf{y}} = J \cdot \dot{\mathbf{x}}$. Thus, we are able to compute the complete Jacobian performing exactly n of these forward sweeps since setting $\dot{\mathbf{x}}_i = \mathbf{e}_i \in I\!R^n$ for $i = 0, \dots, n-1$ gives us successively the n columns of $J \in I\!R^{m \times n}$. So, if $\dot{\mathbf{x}}_i = \mathbf{e}_i$ then $J \cdot \dot{\mathbf{x}}$ computes the partial derivatives of the m dependent variables $y_0, \dots, y_{m-1}$ with respect to $v_{1-n+i} = x_i$. $\mathbf{e}_i$ $(i = 0, \dots, n-1)$ stands for the i-th Cartesian basis vector in $I\!R^n$. Equation (1.3) can be regarded as the so-called **non-incremental** version of the tangent accumulation procedure. Using the abbreviation $a \mathrel{+}= b$ for $a := a + b$ as in `C` we can write it in its **incremental** form as

$$\dot{v}_j \mathrel{+}= c_{jl} \cdot \dot{v}_l \qquad \text{for} \quad j \in S_l \quad \text{and} \quad l = 1-n, \dots, p.$$

Here, it is necessary to initialize $\dot{v}_j = 0$ for $j = 1, \dots, p+m$. However, the incremental approach will play a more important role in the *reverse mode* of AD which we are going to introduce next. This is due to the character of the information involved in the computation, which may be either local or global.

We use bold letters for vectors to be able to distinguish them from scalar values. Hence, $\dot{\mathbf{x}}_i$ denotes one possible initialization for calculating a directional derivative in forward mode while x_i represents the i-th component of the argument vector $\mathbf{x}$. Matrices will be denoted by capital letters.

Reverse mode

A different way to apply the chain rule is exploited in the (scalar) **reverse mode** of AD. Introducing the notation

$$\bar{v}_l \equiv \frac{\partial y_i}{\partial v_l}$$

we get Equation (1.4) which describes the backward propagation of gradients in the reverse mode:

$$\bar{v}_l = \sum_{j \in S_l} c_{jl} \cdot \bar{v}_j \qquad \text{for} \quad l = p, \ldots, 1-n. \tag{1.4}$$

Here, the chain rule is applied to the so-called **adjoints** $\bar{v}_l$. After initializing $\bar{\mathbf{y}}$, one reverse sweep gives us $\bar{\mathbf{x}} = \bar{\mathbf{y}}^T \cdot J$. So, we can compute the complete Jacobian by performing m reverse sweeps, since setting $\bar{\mathbf{y}}_i = \mathbf{e}_i \in I\!R^m$ for $i = 0, \ldots, m-1$ gives us the m rows of J. If $\bar{\mathbf{y}}_i = \mathbf{e}_i$ then $\bar{\mathbf{y}}^T \cdot J$ computes the partial derivatives of y_i with respect to the n independent variables $x_0, \ldots, x_{n-1}$.

Equation (1.4) yields the non-incremental version of the adjoint evaluation procedure. Rather than using this method the accumulation of the adjoints can also be regarded as an incremental approach:

$$\bar{v}_l \mathrel{+}= c_{jl} \cdot \bar{v}_j \qquad \text{for} \quad l \in P_j \quad \text{and} \quad j = p+m, \ldots, 1 \tag{1.5}$$

where we assume that the adjoints $\bar{v}_l$ have been initialized to zero for $l = p, \ldots, 1-n$. The main difference between the two methods is that the non-incremental approach requires global information whereas the incremental does not. In a c-graph context we can think of the further as "sitting" at the vertex associated with v_l and collecting the contributions to its adjoint value from its successors. Thus, we have to look at a whole set of elemental functions φ_j $(j \in S_l)$ at one time, which represents some sort of global information. However, in the incremental case we "sit" at one of the v_j and compute its contribution to the adjoint values of all its predecessors in the c-graph. This can be done by dealing with just one elemental function φ_j at one step, making the required information local. Notice that in the forward mode of AD it is the other way around. Here, the incremental approach requires global information whereas the non-incremental does not.

Considering the scalar mode of AD applied to a scalar c-graph CG we can associate scalar *derivative values* $\tilde{v}_j$ with the vertices $v_j \in CG$. In particular we have $\tilde{v}_j \equiv \dot{v}_j$ in forward mode and $\tilde{v}_j \equiv \bar{v}_j$ in reverse mode. Later we will distinguish between different types of derivative values depending on both the mode and the type of the graph (see Section 1.3).

Vector Mode of AD

Rather then accumulating the Jacobian by performing n subsequent scalar forward sweeps [m scalar reverse sweeps] we could also propagate bundles of $q \geq 1$ tangents [gradients] during one forward [reverse] sweep. This is equivalent to computing the matrix product

$$(I\!R^{m \times q} \ni) \;\; \dot{Y} = \nabla_{\mathbf{x}} F(\mathbf{x}) \, \dot{X} \qquad \left[(I\!R^{n \times q} \ni) \;\; \bar{X} = \bar{Y}^T \nabla_{\mathbf{x}} F(\mathbf{x}) \right].$$

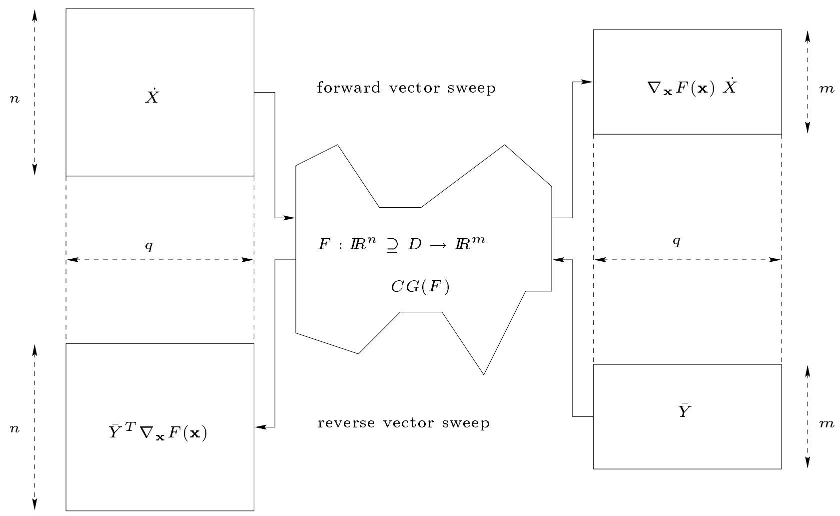

Figure 1.2: Vector Mode of AD

If we set $q = n$ [$q = m$] and initialize $\dot{X}$ [$\bar{Y}$] as the corresponding identity matrix we can apply the local elementary partial derivatives to whole vectors, thus ending up with just one forward [reverse] sweep. This is what we will refer to as the forward [reverse] vector mode which is illustrated by Figure 1.2. The **forward vector mode** of AD can be described by

$$\dot{\mathbf{v}}_j = \sum_{l \in P_j} c_{jl} \dot{\mathbf{v}}_l \qquad \text{for} \quad j = 1, \ldots, p+m \quad \text{(non-incremental)}$$

$$\text{or} \quad \dot{\mathbf{v}}_j {+}{=}\, c_{jl} \dot{\mathbf{v}}_l \quad \text{for} \quad j \in S_l \quad \text{and} \quad l = 1-n, \ldots, p \quad \text{(incremental)},$$

which is equivalent to the scalar version with the local partial derivatives $c_{jl} \in I\!R$ applied to vectors $\dot{\mathbf{v}}_l \in I\!R^q$ in form of element-wise multiplications. As in the scalar mode it is possible to interpret the **reverse vector mode** of AD in two different ways, too. With the adjoints $\bar{\mathbf{v}}_l$ representing row vectors we get as a "recipe" for running the incremental vector adjoint evaluation procedure

$$\bar{\mathbf{v}}_l \mathrel{+}= \bar{\mathbf{v}}_j c_{jl} \qquad \text{for} \quad l \in P_j \quad \text{and} \quad j = p+m, \ldots, 1.$$

The non-incremental vector adjoint evaluation procedure is given by

$$\bar{\mathbf{v}}_l = \sum_{j \in S_l} \bar{\mathbf{v}}_j c_{jl} \qquad \text{for} \quad l = p, \ldots, 1-n.$$

Example

To support the above discussion let us have a closer look at a very simple example in form of a function, given as the following system of non-linear equations:

$$y_0 = \cos(\sin(x_0 \cdot x_1)),$$
$$y_1 = \log(x_0 \cdot x_1).$$

The principle of what happens during both forward and reverse modes is sketched in Figure 1.3. In the first column the initial forward sweep to compute the function value and the local partial derivatives is illustrated. The single assignment code contains two intermediate variables v_1 and v_2 represented by the corresponding two intermediate vertices in the c-graph which is shown in Figure 1.4. In parallel with the evaluation of the scalar assignments the local partial derivatives c_{jl} representing labels on the edges in the c-graph are computed. The second and the third columns list the elementary operations which are performed during the computation of directional derivatives by either one forward sweep or one reverse sweep, respectively.

Function value + partials	**Forward mode**	**Reverse mode**
Evaluation trace	$\dot{v}_j = \sum_{l \in P_j} c_{jl} \cdot \dot{v}_l$	$\bar{v}_l = \sum_{j \in S_l} c_{jl} \cdot \bar{v}_j$
$v_1 = x_0 \cdot x_1 \;\; (c_{1,-1} = x_1,\; c_{1,0} = x_0)$ $v_2 = \sin v_1 \;\; (c_{2,1} = \cos v_1)$ $y_0 = \cos v_2 \;\; (c_{3,2} = -\sin v_2)$ $y_1 = \log v_1 \;\; (c_{4,1} = 1/v_1)$	$\dot{v}_1 = c_{1,-1} \cdot \dot{x}_0 + c_{1,0} \cdot \dot{x}_1$ $\dot{v}_2 = c_{2,1} \cdot \dot{v}_1$ $\dot{y}_0 = c_{3,2} \cdot \dot{v}_2$ $\dot{y}_1 = c_{4,1} \cdot \dot{v}_1$	$\bar{v}_2 = c_{3,2} \cdot \bar{y}_0$ $\bar{v}_1 = c_{2,1} \cdot \bar{v}_2 + c_{4,1} \cdot \bar{y}_1$ $\bar{x}_1 = c_{1,0} \cdot \bar{v}_1$ $\bar{x}_0 = c_{1,-1} \cdot \bar{v}_1$

Figure 1.3: Forward and Reverse Modes of AD (non-incremental)

Notice that in forward mode the computation of the function value and the accumulation of the derivative values can be performed in parallel. This becomes clear when reading the columns that show the computation of the local partial derivatives and the forward propagation of a directional derivative in forward mode, respectively, row by row. Whenever the value of a partial derivative c_{jl} of an intermediate variable v_j with respect to any of its arguments is needed for evaluating its tangent component $\dot{v}_j$ it has already been made available by the corresponding operation in the initial forward sweep. For the reverse mode we need to run an initial forward sweep followed by the reverse sweep to propagate a particular gradient, since the local partial derivatives are used in reverse order. It takes two forward sweeps with $\dot{\mathbf{x}} = \mathbf{e}_0$ and $\dot{\mathbf{x}} = \mathbf{e}_1$, resulting in 8 multiplications, and also two reverse sweeps with $\bar{\mathbf{y}} = \mathbf{e}_0$ and $\bar{\mathbf{y}} = \mathbf{e}_1$ which need only 7 multiplications for calculating the complete Jacobian, provided that we do not count multiplications with zero as a result of the full exploitation of the sparsity structure of the problem. Equivalently, we could do the job in vector mode performing one forward or one reverse sweep with $\dot{X}$ or $\bar{Y}$ initialized as the identity matrix $I \in I\!R^{2\times 2}$, respectively.

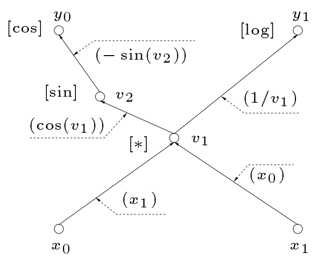

Figure 1.4: Example

Although the values of the Jacobians computed using forward or reverse mode are identical there may be a difference in the computational complexity. This is the motivation for us to study different orders of the application of the chain rule while making full use of the sparsity of a problem. We will introduce the combinatorial optimization problem arising from this discrepancy in Chapter 2.

1.2.2 Edge elimination in c-graphs

The elimination of edges represents the elemental action that we will build on in our approach. It can be regarded as the equivalent to the chain rule when looking at evaluation routines in form of c-graphs.

During our discussion of edge elimination we will distinguish between two situations:

1. The edge (i, j) is eliminated for calculating a contribution to the tangent value $\dot{v}_j$, associated with its target.

2. The elimination of (i, j) is part of the computation of the adjoint value $\bar{v}_i$ of its source.

Forward edge elimination

Consider the following representation of the chain rule:

$$\dot{v}_j = \sum_{l \in P_j} c_{jl} \cdot \dot{v}_l \qquad \text{for} \quad j = 1, \ldots, p + m. \tag{1.6}$$

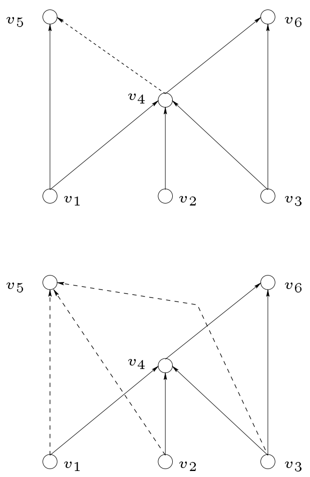

Figure 1.5: $(4, 5)$

Suppose we want to eliminate the term for a particular intermediate vertex v_i from this recurrence, which is equivalent to eliminating the corresponding edge (i, j) from the c-graph. This edge elimination may be regarded as a *forward* elimination in the sense that (i, j) is eliminated as part of the computation of $\dot{v}_j$ representing the derivative value of its target. Our discussion will be illustrated by Figure 1.5 where the elimination of $(4, 5)$ is illustrated.

We start with the isolation of the term that contains the factor $\dot{v}_i$:

$$\dot{v}_j = c_{ji} \cdot \dot{v}_i + \sum_{i \neq l \in P_j} c_{jl} \cdot \dot{v}_l.$$

Then we substitute the expression corresponding to $\dot{v}_i$ by Equation (1.6):

$$\dot{v}_j = \sum_{l \in P_i} c_{ji} \cdot c_{il} \cdot \dot{v}_l + \sum_{i \neq l \in P_j} c_{jl} \cdot \dot{v}_l.$$

Thus, we get a new calculation rule for $\dot{v}_j$ without any terms containing $\dot{v}_i$:

$$\dot{v}_j = \sum_{l \in \tilde{P}_j} \tilde{c}_{jl} \cdot \dot{v}_l \qquad \text{where} \qquad \tilde{P}_j \equiv P_i \cup P_j \setminus \{i\}$$

and

$$\tilde{c}_{jl} \equiv c_{jl} + c_{ji} \cdot c_{il}. \tag{1.7}$$

Equation (1.7) determines the way in which the labels on the edges in the c-graph have to be updated. 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).

In order to be able to distinguish between edges which can be subject to either forward or backward elimination (backward elimination of edges in c-graphs will be introduced below) and those that cannot we introduce the following terminology: An edge (i, j) is called **forward [backward] final** if $v_i \in X$ $[v_j \in Y]$. It is final in the sense that once it occurred in the c-graph (e.g. as the result of the elimination of another edge) it will not be possible to eliminate it forward [backward] again. An edge is called **final** if it is both forward and backward final. An incrementation of the local partial derivative that labels a final edge can still take place as long as there are other paths apart from (i, j) connecting $v_i \in X$ with $v_j \in Y$. If (i, j) is not final it will also be referred to as an **intermediate edge**.

With the above we observe that forward elimination of edges is exactly what happens during the non-incremental forward propagation of tangents in the forward mode of AD. Computing $\dot{v}_j$ is equivalent to the successive forward elimination of all intermediate edges leading into v_j. This process will terminate as soon as all these edges have a minimal source $x_i \in X$, i.e. when they are forward final. At this stage the labels on these edges are exactly the partial derivatives of v_j with respect to the corresponding x_i.

Backward edge elimination

According to the chain rule applied to the adjoints in the reverse mode of AD we have

$$\bar{v}_l = \sum_{j \in S_l} c_{jl} \cdot \bar{v}_j \qquad \text{for} \quad l = p, \ldots, 1-n. \tag{1.8}$$

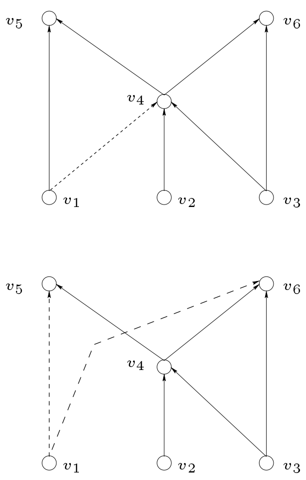

Figure 1.6: $(1, 4)$

Again, we intent to eliminate the term for an intermediate vertex v_i from this recurrence, which is equivalent to the elimination of the corresponding edge (l, i) from the c-graph. This edge elimination may be regarded as a *backward* elimination as it is part of the computation of $\bar{v}_l$, which is always built on the adjoint values of successors of v_l in the c-graph.

Analogous to the forward case we isolate the product involving $\bar{v}_i$

$$\bar{v}_l = c_{il} \cdot \bar{v}_i + \sum_{i \neq j \in S_l} c_{jl} \cdot \bar{v}_j$$

followed by the substitution of the recurrence given for $\bar{v}_i$ by Equation (1.8)

$$\bar{v}_l = \sum_{j \in S_i} c_{il} \cdot c_{ji} \cdot \bar{v}_j + \sum_{i \neq j \in S_l} c_{jl} \cdot \bar{v}_j .$$

This results in a new calculation rule for $\bar{v}_l$ without any terms containing $\bar{v}_i$:

$$\bar{v}_l = \sum_{j \in \tilde{S}_l} \tilde{c}_{jl} \cdot \bar{v}_j \qquad \text{where} \qquad \tilde{S}_l \equiv S_i \cup S_l \setminus \{i\}$$

and

$$\tilde{c}_{jl} \equiv c_{jl} + c_{il} \cdot c_{ji}. \tag{1.9}$$

The graphic interpretation of the backward edge elimination is shown in Figure 1.6. 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. This is what happens during the backward propagation of gradients in the non-incremental reverse mode of AD. Computing $\bar{v}_l$ is equivalent to the successive backward elimination of all intermediate edges emanating from v_l in the c-graph. The process terminates when all these edges have maximal sinks, which makes them backward final.

Whenever the elimination of an edge (i, j) leads to the insertion of a new edge which has not been in the c-graph before we say that fill-in was generated. More general, it is useful to define the **fill-in** as the difference between the number of edges in the c-graph after and before the elimination of (i, j). If an edge which would have to be inserted already does exist then it is said to be **absorbed**. Both the generated fill-in and the concept of absorption will be exploited in Chapter 3 where several heuristics for efficiently applying the chain rule to c-graphs will be presented.

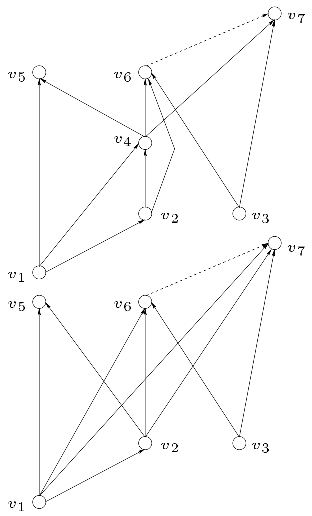

Figure 1.7: v_4

1.2.3 Vertex elimination in c-graphs

Obviously, the forward [backward] elimination of all out-edges [in-edges] of a vertex v_i leads to the elimination of v_i itself. This could be one way to explain vertex elimination. So far, people have approached the problem of calculating Jacobians efficiently by cross-country elimination from the point of view of vertex elimination exclusively. Therefore, we have decided to introduce vertex elimination separately not building on the elimination of edges. This will provide deeper insight into the behavior of the chain rule on c-graphs.

Before we start our discussion of vertex elimination it is necessary to give a few definitions and to introduce the notation. 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.

Considering a given c-graph $CG = (V, E)$ a derivative vector $\tilde{\mathbf{v}}_N \in I\!R^{|V|}$ associated with a vertex set $V' = \{v_i \in V : i \in N\}$ contains the scalar derivative values of all vertices of V'. Furthermore, it is invariant with respect to the projection $I(N)$, i.e. $I(N) \cdot \tilde{\mathbf{v}}_N = \tilde{\mathbf{v}}_N$.

The predecessor [successor] set $P(N)$ $[S(N)]$ of a given set $N = \{i_1, \ldots, i_m\}$ of vertex indices is defined as

$$P(N) \equiv \left(\bigcup_{j \in N} P_j \right) \setminus N \qquad \left[S(N) \equiv \left(\bigcup_{j \in N} S_j \right) \setminus N \right].$$

If $N = \{j\}$ contains only one element we will usually write j and skip the parentheses. In consistency with Section 1.1.2 we set $P(j) \equiv P_j$ $[S(j) \equiv S_j]$. The fact that the predecessor [successor] set of N does not contain elements of N itself is emphasized by the dashed line connecting v_6 and v_7 in Figure 1.7 which is supposed to serve as an illustration for the further development. The vertex v_j which we are about to eliminate can be associated with v_4 in the picture.

Now, we can start the derivation of a rule for the elimination of vertices from the c-graph with the vector-matrix product representation of the chain rule. Say we want to eliminate a particular intermediate vertex v_j. Consider

$$I\!R^{|V|} \ni \quad \dot{\mathbf{v}}_{S_j} = C_{S_j} \cdot \dot{\mathbf{v}}_{P(S_j)}. \tag{1.10}$$

Separating the expression for v_j from Equation (1.10) is equivalent to dividing C_{S_j} into one part that contains all scalar local partial derivatives associated with v_j and a second part which consists of the remaining sensitivities of the v_k, $k \in S_j$, with respect to vertices preceding them in the c-graph:

$$\dot{\mathbf{v}}_{S_j} = \left(C_{S_j} \cdot I(j) + C_{S_j} \cdot I(P(S_j) \setminus j)\right) \cdot \dot{\mathbf{v}}_{P(S_j)}.$$

Notice that the projection of C_{S_j} onto its j-th column yields exactly the local extended adjoint Jacobian of v_j. Substitution of A for $C_{S_j} \cdot I(P(S_j) \setminus j)$ leads to

$$\dot{\mathbf{v}}_{S_j} = (\bar{C}_j + A) \cdot (\dot{\mathbf{v}}_j + \dot{\mathbf{v}}_{P(S_j)\setminus j})$$

where the derivative vector corresponding to the vertex v_j has been isolated from $\dot{\mathbf{v}}_{P(S_j)}$. If we denote the vector containing the derivative values of the remaining vertices v_i, $i \in P(S_j) \setminus j$, by $\mathbf{b}$ we get

$$\dot{\mathbf{v}}_{S_j} = \bar{C}_j \cdot \dot{\mathbf{v}}_j + A \cdot \mathbf{b}.$$

The application of the chain rule to $\dot{\mathbf{v}}_j$ results in

$$\dot{\mathbf{v}}_{S_j} = \bar{C}_j \cdot C_j \cdot \dot{\mathbf{v}}_{P_j} + A \cdot \mathbf{b}$$

and we finally get a new expression for the computation of $\dot{\mathbf{v}}_{S_j}$, which does not contain any dependency on the corresponding scalar value $\dot{v}_j$:

$$\dot{\mathbf{v}}_{S_j} = D \cdot \dot{\mathbf{v}}_{\tilde{P}} \tag{1.11}$$

where

$$D = \bar{C}_j \cdot C_j + A$$

and

$$\tilde{P} = (P(S_j) \cup P_j) \setminus j.$$

Analogous observations can be made based on the matrix product representation of the chain rule as interpreted in the reverse mode of AD. Starting with

$$I\!R^{|V|} \ni \quad \bar{\mathbf{v}}_{P_j} = \bar{\mathbf{v}}^T_{S(P_j)} \cdot \bar{C}_{P_j}$$

and following an argumentation similar to the one developed above we get

$$\begin{aligned}
\bar{\mathbf{v}}_{P_j} &= \bar{\mathbf{v}}^T_{S(P_j)} \cdot \left(I(j) \cdot \bar{C}_{P_j} + I(S(P_j) \setminus j) \cdot \bar{C}_{P(S_j)\setminus j}\right) \\
&= (\bar{\mathbf{v}}^T_j + \bar{\mathbf{v}}^T_{S(P_j)\setminus j}) \cdot (C_j + \bar{A}) \\
&= \bar{\mathbf{v}}^T_j \cdot C_j + \bar{\mathbf{b}} \cdot \bar{A} \\
&= \bar{\mathbf{v}}^T_{S_j} \cdot \bar{C}_j \cdot C_j + \bar{\mathbf{b}} \cdot \bar{A}.
\end{aligned}$$

We have substituted $\bar{A} := I(S(P_j) \setminus j) \cdot \bar{C}_{P(S_j)\setminus j}$ and $\bar{\mathbf{b}} := \bar{\mathbf{v}}^T_{S(P_j)\setminus j}$. Thus, we have derived a new rule for the calculation of $\bar{\mathbf{v}}_{P_j}$, which does not depend on the scalar adjoint $\bar{v}_j$:

$$\bar{\mathbf{v}}_{P_j} = \bar{\mathbf{v}}^T_{\tilde{S}} \cdot \bar{D} \tag{1.12}$$

where

$$\bar{D} = \bar{C}_j \cdot C_j + \bar{A}$$

and

$$\tilde{S} = (S(P_j) \cup S_j) \setminus j.$$

1.2.4 Conclusions

We have observed that the non-incremental forward [reverse] mode of AD can be associated with the successive forward [backward] elimination of all edges leading into [emanating from] v_j that do not have a minimal source for $j = l, \ldots, p+m$ [$j = u, \ldots, 1-n$]. Here, l [u] stands for the lowest [highest] index of any vertex in the c-graph having at least one in-edge [out-edge] with an intermediate source [target]. Analogous, we can consider the incremental versions of the two basic modes of AD in terms of edge eliminations in the c-graph. The incremental reverse mode is equivalent to the backward elimination of all in-edges of v_j with $j = p, \ldots, 1$. Similarly, the incremental forward mode can be described as the forward elimination of all out-edges of vertices v_j with $j = 1, \ldots, p$. Notice that the incremental forward [reverse] mode is equivalent to the elimination of the intermediate vertices v_j in the c-graph for j in ascending [descending] order (see Section 2.3).

The elimination of an intermediate vertex v_j involves exactly

$$|P_j| \cdot |S_j| \equiv \mathbf{OPS}\{v_j\}$$

multiplications each one possibly followed by an addition. The above product, which we will refer to as the **Markovitz degree** of the vertex v_j, is a characteristic value for every vertex describing the cost of eliminating it from the c-graph. We have borrowed this term from the theory of solving large sparse systems of linear equations where it occurs in connection with the minimization of fill-in. Analogous, we observe that the forward [backward] elimination of an edge (i, j) takes

$$|P_i| \equiv \mathbf{OPS_F}\{(i,j)\} \quad [|S_j| \equiv \mathbf{OPS_B}\{(i,j)\}]$$

multiplications, again, each of them possibly followed by an addition. This yields the definition of the **forward [backward] Markovitz degree** of an edge. At the moment the [forward,backward] Markovitz degree appears to be static. Its meaning will be extended in Chapter 2.

Looking at Equations (1.11) and (1.12) let $S(P_j) = S_j$ and $P(S_j) = P_j$. It follows that $\tilde{P} = P_j$ and $\tilde{S} = S_j$. Consider $A = C_{S_j} \cdot I(P(S_j) \setminus j)$, which is now equal to $C_{S_j} \cdot I(S_j)$, and, analogous, $\bar{A} = I(S(P_j) \setminus j) \cdot \bar{C}_{P(S_j)\setminus j}$, which becomes $I(S_j) \cdot \bar{C}_{P_j}$. Both A and $\bar{A}$ contain all local partial derivatives labeling the edges leading from the vertices in P_j to those in S_j, i.e. we have

$$A = \bar{A} \quad \Rightarrow \quad D = \bar{D}.$$

Certainly, this is what we have expected and the recursive application of the above vertex elimination procedure leads to the familiar equations

$$\mathbb{R}^m \ni \quad \dot{\mathbf{y}} = J \cdot \dot{\mathbf{x}} \qquad \text{(forward mode of AD)}, \tag{1.13}$$

$$\mathbb{R}^n \ni \quad \bar{\mathbf{x}} = \bar{\mathbf{y}}^T \cdot J \qquad \text{(reverse mode of AD)}. \tag{1.14}$$

Thus, we have derived a formalism which describes the equivalents to both forward and reverse modes of AD in terms of the elimination of all intermediate vertices in the c-graph. As above, vertex elimination may be regarded as a special edge elimination procedure. The elimination of a vertex v_j is equivalent to any combination of backward eliminating its in-edges and forward eliminating its out-edges, provided this sequence leads to the elimination of v_j before we deal with any other intermediate vertex in the c-graph.

The global extended Jacobian for a single assignment code evaluated at a given argument point is defined as $\mathbb{R}^{|V|\times|V|} \ni J_e = (c_{ji})$, where the c_{ji} are the local elementary partial derivatives labeling the edges which connect v_i and v_j in the c-graph. Obviously, J_e is a square lower triangular matrix. For practical reasons we also define the diagonal elements to be zero, which gives us

$$J_e = \sum_{i=1}^{p+m} C_i = \sum_{j=1-n}^{p} \bar{C}_j$$

as the local extended Jacobian of the vertex set $\{v_i : i \in Z \cup Y\}$ or as the local extended adjoint Jacobian of $\{v_j : j \in X \cup Z\}$.

The process of calculating the complete Jacobian by a combined vertex-edge elimination procedure may be regarded as a transformation process of the extended Jacobian. There is an equivalent for every action in the c-graph in terms of a transformation of J_e:

- Forward elimination of an edge (i, j):

$$\begin{aligned} J_e :&= J_e + \mathbf{e}_j \mathbf{e}_j^T J_e \mathbf{e}_i \mathbf{e}_i^T (J_e - I) \\ &= J_e + \mathbf{e}_j \mathbf{e}_j^T \bar{C}_i C_i - \mathbf{e}_j \mathbf{e}_j^T J_e \mathbf{e}_i \mathbf{e}_i^T. \end{aligned}$$

- Backward elimination of an edge (i, j):

$$\begin{aligned} J_e :&= J_e + (J_e - I) \mathbf{e}_j \mathbf{e}_j^T J_e \mathbf{e}_i \mathbf{e}_i^T \\ &= J_e + \bar{C}_j C_j \mathbf{e}_i \mathbf{e}_i^T - \mathbf{e}_j \mathbf{e}_j^T J_e \mathbf{e}_i \mathbf{e}_i^T. \end{aligned}$$

- Elimination of a vertex v_i:

$$\begin{aligned} J_e :&= (I - \mathbf{e}_i \mathbf{e}_i^T)(J_e + \bar{C}_i C_i)(I - \mathbf{e}_i \mathbf{e}_i^T) \\ &= (I - \mathbf{e}_i \mathbf{e}_i^T) J_e (I + \mathbf{e}_i \mathbf{e}_i^T J_e)(I - \mathbf{e}_i \mathbf{e}_i^T) \\ &= (I - \mathbf{e}_i \mathbf{e}_i^T)(I + J_e \mathbf{e}_i \mathbf{e}_i^T) J_e (I - \mathbf{e}_i \mathbf{e}_i^T). \end{aligned}$$

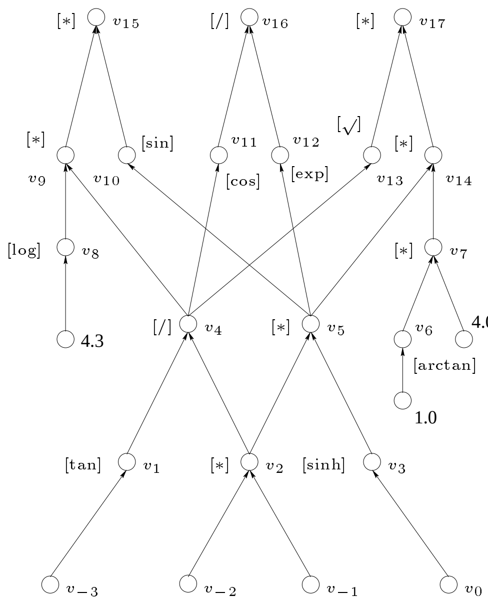

Figure 1.8: Complete Computational Graph

Referring to the forward [reverse] mode of AD we can rewrite Equation (1.13) making use of the local extended Jacobians C_i:

$$\dot{\mathbf{y}} = \left[Q_m(C_{p+m} + I - \mathbf{e}_{p+m}\mathbf{e}_{p+m}^T)\dots(C_1 + I - \mathbf{e}_1\mathbf{e}_1^T)P_n^T\right] \cdot \dot{\mathbf{x}}$$

where P_n and Q_m are the matrices that project a given vector of length $|V|$ to its first n and its last m components, respectively, i.e

$$P_n \equiv [I, 0, \dots, 0] \in I\!R^{n\times|V|} \quad \text{and} \quad Q_m \equiv [0, \dots, 0, I] \in I\!R^{m\times|V|}.$$

By Equation (1.14) we have for the reverse mode $\bar{\mathbf{x}} = \bar{\mathbf{y}}^T \cdot J$, which can also be expressed in terms of the local extended adjoint Jacobians $\bar{C}_i$ delivering

$$\bar{\mathbf{x}} = \bar{\mathbf{y}}^T \cdot \left[Q_m(\bar{C}_p + I - \mathbf{e}_p\mathbf{e}_p^T)\dots(\bar{C}_{1-n} + I - \mathbf{e}_{1-n}\mathbf{e}_{1-n}^T)P_n^T\right].$$

Finally, we have a representation of the complete Jacobian J as two different matrix products given by

$$\begin{aligned} J &= Q_m(C_{p+m} + I - \mathbf{e}_{p+m}\mathbf{e}_{p+m}^T)\dots(C_1 + I - \mathbf{e}_1\mathbf{e}_1^T)P_n^T \\ &= Q_m(\bar{C}_p + I - \mathbf{e}_p\mathbf{e}_p^T)\dots(\bar{C}_{1-n} + I - \mathbf{e}_{1-n}\mathbf{e}_{1-n}^T)P_n^T. \end{aligned}$$

Building on this expression we have developed a dynamic programming algorithm for optimizing the matrix product with respect to the number of scalar multiplications performed. It represents an adaption of the well-known *chained matrix product* dynamic programming algorithm which will be discussed in Chapter 4.

1.2.5 Example

So far, we have discussed how to calculate the entries of the complete Jacobian of a given evaluation routine of a vector function F by converting it into a directed,

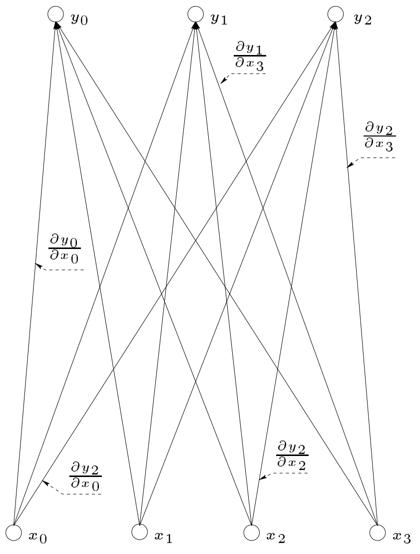

Figure 1.9: $K_{4,3}$

acyclic c-graph followed by the elimination of all intermediate edges [vertices]. Now it is time for an example to illustrate our results.

We will consider the following fragment of a C-program evaluating a vector function

$$F : I\!R^4 \supseteq D \rightarrow I\!R^3 :$$

```
double x[4], y[3];
double h1,h2,pi;
...
h1 = tan(x[0])/(x[1]*x[2]);
h2 = x[1]*x[2]*sinh(x[3]);
pi = 4.*arctan(1.);
y[0] = log(4.3)*h1*sin(h2);
y[1] = cos(h1)/exp(h2);
y[2] = sqrt(h1)*pi*h2;
...
```

It contains the vector $\mathbf{x}$ of the independent variables, the vector $\mathbf{y}$ the components of which represent the dependent variables, and three auxiliary variables h1, h2, and pi. Following the procedure described in Section 1.1 we can break this program down into a sequences of scalar assignment by storing the result of every elemental function in a unique intermediate variable. Thus, we get the following evaluation trace associated with the above evaluation routine:

```
v[1]=tan(v[-3]); v[2]=v[-2]*v[-1]; v[3]=sinh(v[0]); v[4]=v[1]/v[2];
v[5]=v[2]*v[3]; v[6]=arctan(1.); v[7]=4.*v[6]; v[8]=log(4.3);
v[9]=v[8]*v[4]; v[10]=sin(v[5]); v[11]=cos(v[4]); v[12]=exp(v[5]);
v[13]=sqrt(v[4]); v[14]=v[5]*v[7]; v[15]=v[9]*v[10]; v[16]=v[11]/v[12];
v[17]=v[13]*v[14];
```

Building on the single assignment code the complete c-graph CG_F^c displayed in Figure 1.8 can be derived. For a given argument vector $\mathbf{x}$ we are now able to calculate the function value $\mathbf{y}$ and the local partial derivatives c_{ji} labeling the edges in CG_F^c by performing the initial forward sweep. After the removal of all dead vertices, i.e. those intermediate vertices which either are not reachable from any of the minimal vertices or from which none of the maximal vertices can be reached, we get the reduced version of the c-graph with the corresponding extended Jacobian matrix J_e. This is when the interesting part begins. By successively eliminating all intermediate vertices [edges] from the c-graph we will finally reach our goal in the shape of the complete bipartite c-graph $K_{4,3}$ shown in Figure 1.9, which represents the complete Jacobian.

In order to illustrate this process let us have a look at the elimination of v_2 followed by the backward elimination of $(4, 9)$.

$$J_e := (I - \mathbf{e}_2\mathbf{e}_2^T)(J_e + \bar{C}_2 C_2)(I - \mathbf{e}_2\mathbf{e}_2^T)$$

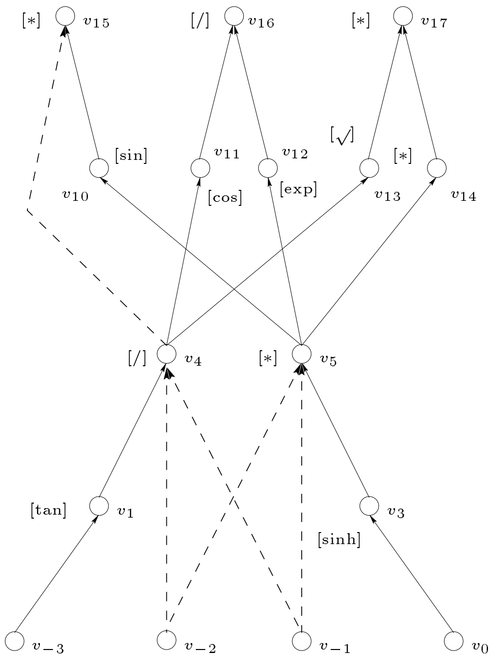

Figure 1.10: Transformed Computational Graph

describes the elimination of v_2 in terms of a transformation of J_e. Taking this new extended Jacobian and performing the equivalent to the backward elimination of $(4, 9)$ results in

$$J_e := J_e + (J_e - I)\mathbf{e}_9\mathbf{e}_9^T J_e \mathbf{e}_4 \mathbf{e}_4^T.$$

Here, the backward elimination of $(4, 9)$ is equivalent to the elimination of v_9. The c-graph after these actions is shown in Figure 1.10, with the new edges highlighted. As illustrated by this example the application of the chain rule to functions, algorithms, or c-graphs is not restricted to the approach exploited in the forward or reverse modes of AD. In Chapter 2 we will deal with arbitrary application orders and we will describe the combinatorial optimization problem arising from this setup.

1.3 Getting More General

So far, we have assumed the derivative objects associated with the vertices in the c-graph to be scalars or a set of independent scalar values in vector mode where we have an element-wise application of the local partial derivatives to the components of a vector. 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 1.3.1 *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*

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

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 1.3.2 *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 1.11. 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 we have referred to as the vector mode of AD up to this point.

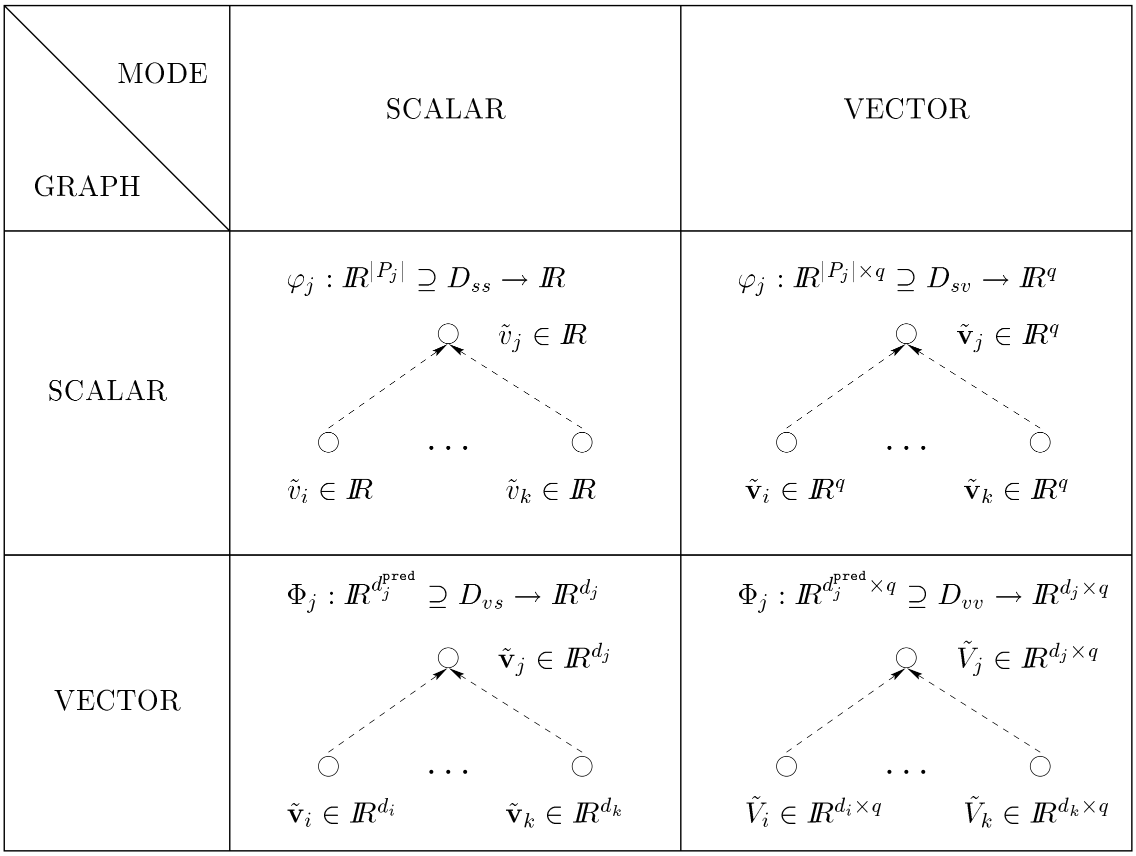

Figure 1.11: Classification

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 1.1.3 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

$$\varphi_j : I\!R^{|P_j|} \supseteq D \to I\!R \qquad \text{for} \quad j = 1, \ldots, p+m$$

with respect to each of their arguments v_i, with $i \in P_j$, 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 which we are going to discuss in Section 1.5. In general vector mode a single assignment code is given by

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

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 forward vector mode (incremental form):

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

General reverse vector mode (incremental form):

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

General reverse vector mode (non-incremental form):

$$(I\!R^{q \times d_j} \supseteq) \;\; \bar{V}_j = \sum_{k \in S_j} \bar{V}_k C_{kj} \qquad \text{for} \quad j = p, \ldots, 1-n.$$

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 $(i,j) \in CG$ and $(j,k) \in CG$ we get the 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

$$\forall i \in P_j : \quad C_{ki} \mathrel{+}= \; C_{kj} \cdot C_{ji}$$

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:

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

Anticipating the beginning of Chapter 2, 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.

$$\forall k \in S_j : \quad C_{ki} \mathrel{+}= \; C_{kj} \cdot C_{ji}.$$

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 1.2.4 we call $\mathbf{OPS_F}\,\{(i,j)\}$ [$\mathbf{OPS_B}\,\{(i,j)\}$] the **forward [backward] Markovitz 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 **Markovitz degree** of v_j in the scalar mode if $d_j = 1$ for all j.

1.4 Computational Line Graphs

The **line graph** $LG = (E, E')$ of a c-graph $CG = (V, E)$ has the edges of CG as its vertices and a vertex $(i_1, j_1) \in LG$ is adjacent to [from] $(i_2, j_2) \in LG$ whenever

$$\exists \;\; v \in CG : \quad v_{j_1} = v = v_{i_2} \quad [v_{j_2} = v = v_{i_1}].$$

Furthermore, its minimal [maximal] vertices correspond to assumed unique in-edges [out-edges] leading into [emanating from] the minimal [maximal] vertices in CG. LG is acyclic as a cycle in LG immediately results in a cycle in CG, which would lead to a contradiction to CG being acyclic. The edges of LG can be partitioned into complete bipartite subgraphs in such a way that no vertex is contained in more than two of the subgraphs.

The shortest semi-cycle in LG has length five. It follows that two successors of a given vertex in LG cannot have a common successor. Analogous, any two predecessors of a vertex in LG cannot have a common predecessor.

It is straight forward to formulate rules for both forward and backward elimination of vertices in LG based on the corresponding elimination rules in CG (see Figure 1.12). However, suppose we want to apply a rule similar to the one used for vertex elimination in CG to the vertices in LG. Obviously, we must adapt the vertex elimination rule in a way that it will be applicable to the line graph. Notice, that above all it should be such that the elimination of a vertex in LG will transform it in a way that the result is again a line graph. We denote the set of vertices in LG having a distance of two (i.e. there is an edge path of length two) *to* (i,j) by $P^2_{(i,j)} \subseteq E$ and, analogous, the set of vertices in LG with the same distance *from* (i,j) by $S^2_{(i,j)} \subseteq E$. Basically, in CG the elimination of (i,j) is equivalent to its forward elimination followed by the backward elimination of all edges connecting the vertices from the predecessor set of its source v_i with its target. We will refer to this method as the **forward-backward approach**. Alternatively, we could also choose to eliminate (i,j) backward followed by the forward elimination of all edges leading from v_i into the successor set of v_j (**backward-forward approach**). Thus, we have two methods for eliminating a vertex from LG which are equivalent with respect to the transformation they induce on LG. However, they may be different regarding their costs. Denoting the cardinality of $P^2_{(i,j)}$ $[S^2_{(i,j)}]$ by $|P^2_{(i,j)}|$ $[|S^2_{(i,j)}|]$ we observe that

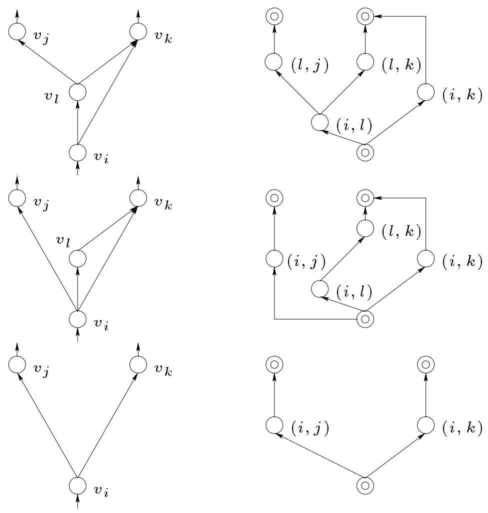

Figure 1.12: $CG \leftrightarrow LG$

$$\mathbf{OPS_{F-B}}\{(i,j)\} \equiv |P^2_{(i,j)}| + |P^2_{(i,j)}||S^2_{(i,j)}|$$

multiplications are required for the forward-backward approach, whereas the backward-forward version takes

$$\mathbf{OPS_{B-F}}\{(i,j)\} \equiv |S^2_{(i,j)}| + |P^2_{(i,j)}||S^2_{(i,j)}|$$

multiplications.

Computational line graphs provide a different perspective on numerous problems associated with c-graphs. In some cases it may be advantageous to consider line graphs if we wish to visualize certain aspects of the underlying c-graph. However, there is always an equivalent formulation for each statement on line graphs in terms of the original graph. Therefore, we will leave it at this short introduction.

1.5 Hierarchical Application of the Chain Rule

The hierarchical approach to the application of the chain rule to c-graphs can be regarded as a direct consequence from the definition of vector c-graphs. Instead of breaking the evaluation routine down into scalar valued elemental functions we allow the latter to be vector functions themselves. This enables us to 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 even make sense if the chain rule is not applied in the cross-country mode but in the basic forward and reverse modes of AD.

We call subgraphs of c-graphs L_0-graphs if they consist of vertices that represent unary and binary elementary operations exclusively. Then the hierarchical approach leads to L_j-graphs with $j > 0$ representing higher levels of abstraction of the original c-graph. In general, the L_0-graphs will be scalar graphs as they are derived directly from parts of the underlying evaluation routine. Code which is called repeatedly within a loop (*loop body function*) or subroutine and function calls are potential candidates for separate vertices at some level L_j of abstraction. In this case they represent vector functions the c-graphs of which are L_{j-1}-graphs.

Building on the results from Section 1.3 the chain rule can be applied to any level separately. In Chapter 2 we will discuss the combinatorial optimization problem which arises from the objective to minimize the number of multiplications required to accumulate the complete Jacobian of the vector function which is given as a c-graph. We will propose heuristics and other methods for approximating optimal edge [vertex] elimination sequences. Most of them will have a time complexity which is polynomial in the number of intermediate edges [vertices] in the c-graph. They will work well on small problems with only a few hundred vertices. However, their run-time increases dramatically when applied to large-scale problems. The hierarchical approach is the only way to overcome this problem.Therefore, it will play an important role in the implementation of our results.

Chapter 2

Optimizing the Application of the Chain Rule

2.1 A Shortest Path Problem

The Jacobian matrix of a vector function defines a linear transformation on real vectors. We will pick one particular representative J associated with a given vector function F and think about a way to compute all its entries at a minimal cost. 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 $\mathbf{Cost}\{J\}$.

Apart from a large number of multiplications the calculation of the complete Jacobian involves a varying number of additions, too. Modern computers are able to perform additions and multiplications in the same time. So, we could think about not only minimizing the number of multiplications but also the number of additions or even the sum of both. We have the impression that the effort would not pay off since the number of additions is in general small compared to the number of multiplications. Therefore, we will concentrate on the problem of minimizing the number of multiplications by determining nearly optimal edge and vertex elimination orders.

Undoubtedly, it will not be sufficient to think about operations counts while trying to make the computation of the complete Jacobian as fast as possible – and this is what we actually want. Memory accesses will play an important role as they usually dominate the overall execution time on the computer. However, this problem should be examined as part of a different project and therefore we will not go into detail with it. The final objective must be to merge the results from both projects in order to generate optimized derivative code as a compromise between the minimal number of arithmetic operations and an optimal memory access pattern (see Chapter 7).

Building on work of Rose and Tarjan [RoTa78] Herley showed in an unpublished paper that the computation of a vertex elimination sequence that minimizes the fill-in in the c-graph of a vector function is an NP-hard combinatorial optimization problem. We expect the same property to hold for the closely related problem of minimizing the number of multiplications required to accumulate the complete Jacobian matrix although there is no proof for it, so far.

The successive elimination of edges from the c-graph defines the so-called **metagraph** (also general metagraph)

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

the vertices of which represent all different c-graphs that could possibly be derived from CG by edge elimination. Sometimes we will call the vertices $w_k \in V_M$ of the metagraph **stages**, thus indicating that they represent certain intermediate forms on our way from the original c-graph to a subgraph of the complete bipartite graph $K_{n,m}$ which represents the complete Jacobian. If there are $p \gg \max\{m, n\}$ intermediate vertices in CG, which can be connected by at most $\binom{p}{2}$ edges each of them either being in the graph or not, then an upper bound for the number of vertices in M is given by $2^{\binom{p}{2}}$. This value which is not very useful for practical situations can be improved if we take into account that certainly there are vertices in CG which are mutually unreachable. These vertices will never be connected by an edge regardless of the chosen edge elimination sequence. By denoting the number of edges in the transitive closure of CG by E^* we get $2^{|E^*|}$ as a tighter upper bound for the number of vertices in the metagraph. 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. This **general edge elimination problem** could be solved using, for example, the algorithm by Bellman and Ford. Its run-time is proportional to $O(|V_M| \cdot |E_M|)$. The difficulties arise from the fact that both the number of vertices and the number of edges in the metagraph depend on the number of intermediate vertices in the original c-graph exponentially. Therefore, an exhaustive search as well as any algorithm for computing a shortest path in M are not practicable.

In Section 1.2 we have introduced the [forward, backward] Markovitz degree of [edges] vertices in CG. Notice that [$\mathbf{OPS_F}\,\{(i,j)\}$, $\mathbf{OPS_B}\,\{(i,j)\}$] $\mathbf{OPS}\,\{v_j\}$ is not static. Its value changes with the ongoing elimination of vertices and edges, which makes our objective to minimize $\mathbf{Cost}\{J\}$ a computationally hard combinatorial optimization problem. In order to be able to give an exact description of the shortest path problem which we intent to solve it is necessary to give precise definitions of what we mean by the [forward, backward] Markovitz degree [edges] vertices at a given stage in the metagraph.

Definition 2.1.1 *Let $w_k \in V_M$ be a stage in the metagraph $M = (V_M, E_M)$ representing the search space of the general edge elimination problem in a given c-graph $CG = (V, E)$. Let $CG_k = (V_k, E_k)$ be the c-graph associated with w_k.*

We call the number of multiplications required to eliminate an edge $(i, j) \in E_k$ forward [backward] the **forward [backward] Markovitz degree of** (i, j) **at the stage** $w_k \in V_M$ *and we denote it by*

$$\mathbf{OPS_F}^k\,\{(i,j)\} \qquad \left[\quad \mathbf{OPS_B}^k\,\{(i,j)\}\quad\right].$$

Furthermore, we introduce

$$\mathbf{OPS}_{[\mathbf{F},\mathbf{B}]}{}^{k}\left\{(i,j)\right\} \;\equiv\; \min\left\{\mathbf{OPS}_{\mathbf{F}}{}^{k}\left\{(i,j)\right\}, \mathbf{OPS}_{\mathbf{B}}{}^{k}\left\{(i,j)\right\}\right\}.$$

For a vertex $v_j \in V_k$ *we define its* **Markovitz degree at the stage** $w_k \in V_M$ *as the number of multiplications involved in its elimination from* CG_k *and denote it by*

$$\mathbf{OPS}^{k}\left\{v_j\right\}.$$

The minimal number of multiplications required to convert the original c-graph CG_0 *into the graph* CG_k *by the [forward, backward] elimination of [edges] vertices will be referred to as the* **overall Markovitz degree** *at stage* w_k*.*

For the scalar mode on a scalar c-graph we get

$$\mathbf{OPS}_{\mathbf{F}}{}^{k}\left\{(i,j)\right\} = |P_i|_k, \quad \mathbf{OPS}_{\mathbf{B}}{}^{k}\left\{(i,j)\right\} = |S_j|_k, \quad \text{and} \quad \mathbf{OPS}^{k}\left\{v_j\right\} = |P_j|_k \cdot |S_j|_k$$

where $|P_j|_k$ $[|S_j|_k]$ denotes the number of predecessors [successors] of v_i $[v_j]$ in CG_k. Every stage w_k in the metagraph can be associated with a pair (m_k, ES_k) consisting of the overall Markovitz degree which is equal to the shortest path from the source w_0 of the metagraph to w_k and a sequence ES_k of labeled edges. The latter implies the order in which the edges were eliminated such that the number of multiplications involved in this process is equal to m_k, which is locally minimal. The labels indicate whether an edge $(i,j) \in E$ was eliminated forward or backward, i.e.

$$(i,j)_{[\mathbf{F},\mathbf{B}]} \equiv \begin{cases} (i,j)_{\mathbf{F}} & \text{for forward elimination of } (i,j) \\ (i,j)_{\mathbf{B}} & \text{for backward elimination of } (i,j) \end{cases}.$$

The initial stage $w_0 \in V_M$ represents the original c-graph with none of its intermediate edges eliminated. It is uniquely defined by $(m_0 = 0, ES_0 = ())$. An edge connects a stage $w_k \in V_M$ with another stage $w_l \in V_M$ if there is an edge $(i,j) \in CG_k$ the [forward, backward] elimination of which leads to CG_l, i.e.

$$(k,l) \in E_M$$

$$\Leftrightarrow$$

$$\exists (i,j) \in E_k: \qquad ES_l = (ES_k, (i,j)_{[\mathbf{F},\mathbf{B}]}), \quad m_l = m_k + \mathbf{OPS}_{[\mathbf{F},\mathbf{B}]}{}^{k}\{(i,j)\}.$$

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. Let the optimal cost achieved by a general edge elimination method be denoted by $\mathbf{Cost}\{J\}$. Then we have for a given restriction ρ

$$\mathbf{Cost}_\rho\{J\} = c_\rho \cdot \mathbf{Cost}\{J\} \qquad \text{for some} \quad c_\rho \geq 1. \tag{2.1}$$

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. Subsequently, we will discuss some practicable restrictions which lead to significant reductions of the size of M.

1. **Forward edge elimination** ($\rho = \mathbf{F}$)

   Suppose that we permit edges to be eliminated forward exclusively. Having only one elemental action for edge elimination this leads to a subgraph $M_\mathbf{F} \subseteq M$ the size of which has been reduced compared to M. For each stage $w_k \in M$ there are stages adjacent from w_k which can be reached only by the backward elimination of some edge in CG_k. For example, this is the case for those stages $w_s \in M$ where CG_s is derived from CG_k through the elimination of an edge $(i, j) \in E_k$ for which $v_i \in X$ is a minimal vertex, making the forward elimination of (i, j) impossible. The length of a shortest path in $M_\mathbf{F}$ will be denoted by $\mathbf{Cost_F}\{J\}$. Obviously, we can construct special cases where the restriction to pure forward elimination does not lead to a reduction of the size of M. Since these situations are not interesting from the point of view of the optimization of cross-country elimination we ignore them.

   In Chapter 4 we will deal with the representations of the edge elimination problem as a scalar chained matrix product and as a chained matrix product on the so-called *quotient graph*. Both approaches will lead to further reductions of the size of the metagraph leaving us with optimization problems which can be solved in a time that is polynomial in the number of intermediate vertices in the original c-graph. There are versions of the two problems that are based on products of local extended Jacobians such that the corresponding edge eliminations can be regarded as forward.

   A different subgraph of the metagraph is addressed by restricting ourselves to forward edge elimination in the quotient graph which represents a generalization of the corresponding chained matrix product optimization problem. Unfortunately, this problem is exponential in the number of vertices in the quotient graph.

   A detailed discussion of the forward elimination of edges as the result of the product of two local extended Jacobians will be presented in the Sections 4.2 and 4.3.

2. **Backward edge elimination** ($\rho = \mathbf{B}$)

   Analogous statements as for the restriction to forward edge elimination can be made for the backward edge elimination case. The solution of the corresponding shortest path problem in $M_\mathbf{B}$ will be denoted by $\mathbf{Cost_B}\{J\}$. We will also consider the so-called *adjoint* versions of the corresponding chained matrix products. Thus, we get further subgraphs of the metagraph on which the shortest path problem can be solved in a time which is polynomial in the number of vertices in the c-graph. They will be regarded in a dynamic programming context in Chapter 4.

   As in the case of forward edge elimination the backward elimination of edges in a computational quotient graph can be regarded as edge elimination in vector graphs where the originally scalar labels on the edges become matrices containing the local sensitivities of the elements of their target with respect to their source (see Section 1.3).

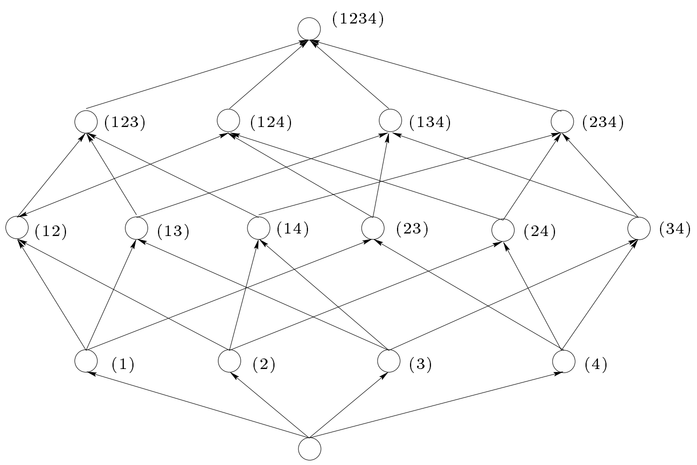

Figure 2.1: Vertex Metagraph $M_{\mathbf{V}}$

3. **Vertex elimination** ($\rho = \mathbf{V}$)

   The restriction to successively eliminating vertices from the c-graph has been exploited in several papers dealing with the minimization of the overall operations count for computing Jacobians ([GrRe91], [Bis96]). There are exactly $p!$ different elimination orders for a problem in p intermediate variables. Vertex elimination leads to the so-called **vertex metagraph** $M_{\mathbf{V}} \subseteq M$ an instance of which is shown in Figure 2.1. The minimal cost of computing the Jacobian using an optimal vertex elimination order is equal to the sum of all Markovitz degrees of the intermediate vertices v_j at the stages $w_{i(j)} \in M_{\mathbf{V}}$ of their elimination, i.e.

$$\mathbf{Cost_V}\{J\} = \sum_{1<j\leq p} \mathbf{OPS}^{i(j)}\{v_j\}. \tag{2.2}$$

   The notation $i(j)$ is supposed to emphasize that the stage at which a particular vertex v_j is eliminated depends on the chosen method for determining the elimination order. A cost $\mathbf{OPS}^{i(j)}\{v_j\}$ corresponds to every vertex in CG depending on the stage $w_{i(j)} \in M_{\mathbf{V}}$ at which it is eliminated under the selected method. The problem of minimizing the cost defined by Equation (2.2) by determining an optimal vertex elimination order is called a **vertex elimination problem**. It is equivalent to the solution of a shortest path problem on the subgraph of the metagraph which is induced by the restriction to the elimination of vertices, i.e. the vertex metagraph. For a problem leading to a c-graph that contains p intermediate vertices there are

$$1 + \sum_{i=0}^{p-1} \prod_{j=0}^{i} (p-j)$$

   vertices [stages] in the corresponding vertex metagraph. This makes a naive exhaustive search method impracticable. Some considerable improvement can be achieved by using a dynamic programming approach based on the interpretation of the vertex metagraph as a multistage graph as proposed in [Bis96].

We have to tabulate the solution to the 2^p subproblems corresponding to the vertices. Obviously, the complexity of this algorithm is still exponential in p.

By putting the focus on the vertex elimination problem we will in general not be able to solve our original problem which was to determine a method for accumulating the Jacobian using a minimal number of multiplications. In Section 2.2 we will show that there are situations in which the optimal vertex elimination sequence does in fact not minimize our objective function $\mathbf{Cost}\{J\}$. In order to be able to find this optimum we have to solve the *general edge elimination problem* yielding the possibility to eliminate edges both forward and backward.

The elimination of a vertex in the computational quotient graph can again be regarded as either the forward elimination of all out-edges of a particular *supernode* (see Section 4.3 for the definition) or, equivalently, the backward elimination of its in-edges. Computational quotient graphs are special vector c-graphs. Graphically there is no difference to the vertex elimination in scalar c-graphs. However, according to our observations from Section 1.3 the costs are different.

Analogous to computational quotient graphs the hierarchical approach to the elimination problem in c-graphs leads to vector c-graphs with the labels on the edges being matrices representing local Jacobians. Hierarchical vertex elimination allows us to structure large-scale problems by splitting them into smaller subproblems which can be treated separately (see Section 1.5).

4. **Vertex elimination in the line graph** ($\rho = \mathbf{VL}$)

Vertex elimination in the line graph of CG is equivalent to the forward elimination of an edge in CG followed by the backward elimination of all edges which were generated or the local partial derivatives of which were incremented (*forward-backward approach*). Analogous, it can be regarded as the backward elimination of the corresponding edge followed by the forward elimination of all filled in or incremented edges (*backward-forward approach*). Numerically the result is the same in both cases. However, the costs may differ. According to what we have discussed in Section 1.4 we observe that

$$\mathbf{OPS_{F-B}}^k\{(i,j)\} = |P_i|_k + |P_i|_k \cdot |S_j|_k$$

and

$$\mathbf{OPS_{B-F}}^k\{(i,j)\} = |S_j|_k + |S_j|_k \cdot |P_i|_k.$$

We define the cost of eliminating a vertex in the line graph of the c-graph to be the minimum out of the number of multiplications required for each of the two approaches, i.e.

$$\mathbf{OPS_{VL}}^k\{(i,j)\} = \begin{cases} \mathbf{OPS_{F-B}}^k\{(i,j)\} & \text{if} \quad |P_i|_k \le |S_j|_k \\ \mathbf{OPS_{B-F}}^k\{(i,j)\} & \text{otherwise} \end{cases}.$$

We could ask whether there is some interpretation of the [forward, backward] elimination of single edges in the computational line graph LG. In LG two vertices are connected by an edge if the two corresponding edges in CG are incident. So, the elimination of an edge in LG would lead to a situation where the two edges in CG do still exist but are no longer incident. There is no meaningful explanation for this situation in our context and we will therefore refrain from discussing edge elimination in computational line graphs.

5. **Vertex elimination in the line graph of quotient graph**

   It is possible to apply the above to the line graph of the computational quotient graph leading to another subgraph of the metagraph.

Furthermore, we can select different subgraphs of M by varying the vertex numbering scheme, which leads to new quotient graphs.

Suppose we have a c-graph that contains p_0 intermediate edges. Each of them has to be eliminated either forward or backward. Undoubtedly, it is possible to anticipate the shape of the vertex metagraph a priori whereas we will get into trouble when trying that for the general metagraph. The reason for this is the generated fill-in. The number of edges in the c-graph is not monotonically decreasing during the elimination process and it is impossible to anticipate it without actually performing the eliminations. Let there be p_{i_1} intermediate edges in the c-graph CG_{i_1} at stage w_{i_1} of the metagraph. Suppose we get to w_{i_2} by eliminating (j,k) from CG_{i_1}. Then we have

$$p_{i_2} = \begin{cases} p_{i_1} - 1 + |S_k|_{i_1} - \overline{\varsigma}_{jk} & \text{for } (j,k)_{\mathbf{B}} \text{ and } |P_k|_{i_1} > 1 \\ p_{i_1} - 1 - \overline{c}_{jk} & \text{for } (j,k)_{\mathbf{B}} \text{ and } |P_k|_{i_1} = 1 \\ p_{i_1} - 1 + |P_j|_{i_1} - \underline{\varsigma}_{jk} & \text{for } (j,k)_{\mathbf{F}} \text{ and } |S_j|_{i_1} > 1 \\ p_{i_1} - 1 - \underline{c}_{jk} & \text{for } (j,k)_{\mathbf{F}} \text{ and } |S_j|_{i_1} = 1 \end{cases}$$

where $\underline{\varsigma}_{jk}$ $[\overline{\varsigma}_{jk}]$ denotes the number of common successors [predecessors] of v_j and v_k at stage w_{i_1}.

Example

With the help of the very simple graph shown in Figure 2.3 in the box number 0 we will now have a closer look at the problem of minimizing the number of multiplications required for the accumulation of the Jacobian via the solution of a shortest path problem in the corresponding metagraph. Our example c-graph represents a scalar valued function $F : I\!R^2 \supseteq D \to I\!R$ like for instance

$$y = F(x_0, x_1) = x_0 \sin(x_1) \sin(x_1).$$

But this is not as relevant as the structure of the c-graph. We have one dependent vertex v_3 depending on two independent vertices v_{-1} and v_0 and the complete c-graph consists of five vertices which are connected by five edges. Notice that there is just one intermediate edge that can be eliminated both forward and backward $((1,2))$. Furthermore, we have two edges which are allowed to be eliminated forward only ($(2,3)$ and $(1,3)$) and the same number of edges which have to be eliminated backward ($(-1,2)$ and $(0,1)$).

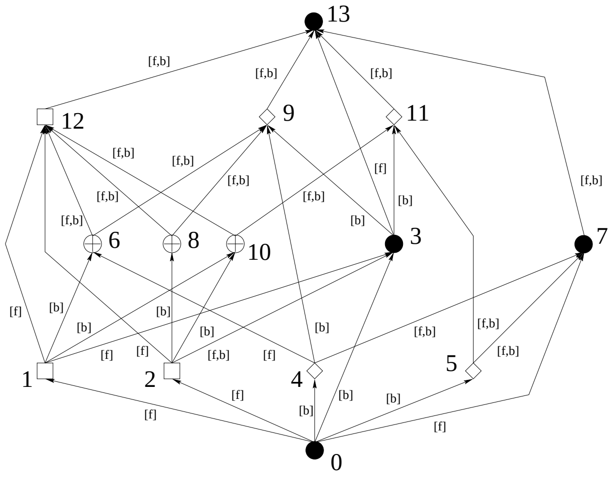

Figure 2.2: Metagraph

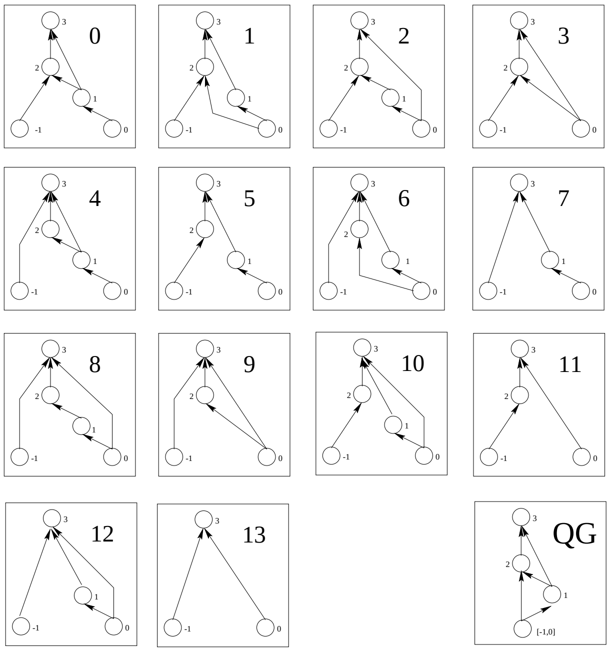

Figure 2.3: Metanodes

The objective is to find a sequence of (forward and backward) edge eliminations which will give us the complete Jacobian in form of the complete bipartite graph $K_{2,1}$ at a minimal cost. We start with listing all intermediate forms of the c-graph that can be constructed using forward and/or backward edge elimination. For this small example we end up with the fourteen different c-graphs displayed in Figure 2.3 including both the original c-graph (0) and the $K_{2,1}$ (13), which represents the complete Jacobian (or in our case the gradient) of the underlying vector function F. These fourteen c-graphs are the vertices of the metagraph $M = (V_M, E_M)$ which is shown in Figure 2.2. Two vertices $w_i, w_j \in V_M$ are connected by an edge $(i, j) \in E_M$ if and only if it is possible to get from the intermediate c-graph CG_i represented by w_i to the one associated with w_j by either forward or backward elimination of an edge in CG_i. We have indicated this fact by labeling the edges (i, j) in M with `[f]` [`[b]`] if we can construct CG_j from CG_i by forward [backward] elimination of an edge in CG_i only. Whenever there are both ways to get from CG_i to CG_j the edges are marked with `[f,b]`. In this case we could regard the corresponding edges as multiple consisting of two parallel edges – one for the forward elimination and the other representing the reverse elimination of an edge in CG_i. This will only be the case for the in- and out-edges of hoisting vertices. Here, both forward elimination of the out-edge as well as the backward elimination of the in-edge take exactly one multiplication. Having in mind that we are about to solve a shortest path problem on the metagraph with the edges representing distances corresponding to the costs of the associated edge eliminations we can consider all parallel edges in the metagraph as equivalent. Thus, we get that the metagraph is a directed, connected, and acyclic graph which does not contain any multiple edges.

We will consider three strategies apart from the general edge elimination method: Forward edge elimination ($\rho = \mathbf{F}$), backward edge elimination ($\rho = \mathbf{B}$), and vertex elimination($\rho = \mathbf{V}$). With every restriction ρ we can associate a certain subgraph

M_ρ	V_{M_ρ}	symbol
$M_{\mathbf{F}}$	$\{0, \mathbf{1}, \mathbf{2}, 7, \mathbf{12}, 3, 13\}$	$\square$
$M_{\mathbf{B}}$	$\{0, 3, \mathbf{4}, \mathbf{5}, \mathbf{9}, \mathbf{11}, 7, 13\}$	$\diamond$
$M_{\mathbf{V}}$	$\{0, 3, 7, 13\}$	$\bullet$

Figure 2.4: M_ρ

$M_\rho \subseteq M$ on which we are going to solve the shortest path problem. Obviously, the source (0) and the sink (13) will be contained within any of the different subgraphs as we always start with the original c-graph and convert it into a subgraph of the complete bipartite graph. Figure 2.4 lists the vertices contained within the subgraphs M_ρ. The vertices corresponding to graphs which can be constructed by using one particular method exclusively are highlighted. In the metagraph we have visualized this fact by different vertex styles. Notice that three vertices (6,8,and 10) cannot be reached using any of the restricted methods and will therefore be contained only within the metagraph representing the general edge elimination strategy. From the numerical point of view the metagraph could contain even more vertices as there may occur situations where two intermediate forms of the c-graph are identical with respect to their structure while carrying different edge labels. Consider, for example, the graph associated with vertex 10 in the metagraph. There are two ways to transform CG_0 by eliminating edges such that we get to a graph with the structure of CG_{10}. We could forward eliminate $(1, 3)$ ($\Rightarrow CG_2$) followed by the backward elimination of $(1, 2)$. Alternatively, the forward elimination of $(1, 2)$ takes us to CG_1 where we would have eliminate $(0, 2)$ backward. Consider the local partial derivative labeling edge $(0, 3)$ in CG_{10}. Choosing the way via CG_1 we have that $c^1_{30} = c_{32}c_{21}c_{10}$. In contrast to that the sequence which derives CG_{10} from CG_2 leads to $c^2_{30} = c_{31}c_{10}$. Assuming that $y = x_0 \sin(x_1) \sin(x_1)$ we would get for the further case $c^1_{30} = \sin(x_1) \cos(x_1) \sin(x_1)$ and for the latter $c^2_{30} = x_0 \sin(x_1) \cos(x_1)$. Naturally, the values of the remaining labels are such that in the end we will get the unique gradient.

2.2 Vertex versus Edge Elimination

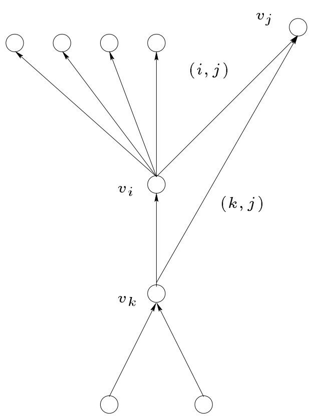

Figure 2.5: Surprise

In the past people used to think about the application of the chain rule to c-graphs 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]). In fact, this is not entirely true since 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 2.5. 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

$$((v_i, v_k) \mathrel{\hat{=}} 5 + 10 =)\ 15 \quad \text{and} \quad ((v_k, v_i) \mathrel{\hat{=}} 4 + 10 =)\ 14$$

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

$$(((i,j), v_k, v_i) \quad \hat{=} \quad 1 + 4 + 8 =) \ \mathbf{13}$$

multiplications which is obviously less than $14 = \min\{(v_i, v_k), (v_k, v_i)\}$ in the pure vertex elimination mode. Thus, we have that any possible vertex elimination sequence will not be able to compute the Jacobian using the minimal number of multiplications. The only way to get the required minimum is to find an optimal edge elimination order. This makes the task even harder as there are always more edges than intermediate vertices in any c-graph. Undoubtedly, the truth lies somewhere between a pure vertex and a pure edge elimination approach.

The fact that vertex elimination is not necessarily optimal gives rise to a fundamental question: *"How bad" is the "best" vertex elimination sequence compared to the "best" edge elimination sequence in general?*

Our goal is to build the *worst case scenario* in order to find out how the minimal Markovitz degree computed by an optimal vertex elimination sequence relates to the actual minimum achieved by applying the optimal edge elimination sequence. Obviously, the sense of any research on vertex elimination would become questionable if $\mathbf{Cost_V}\{J\} = c_\mathbf{V} \cdot \mathbf{Cost}\{J\}$ and the supremum of $c_\mathbf{V}$ is large or even of order $O(p)$ where p is the number of intermediate vertices in CG.

We do not expect this to be the case. This is supported by numerous test results where we have compared some methods for edge elimination with the values delivered by pure vertex elimination. However, the above example is motivation enough for a more detailed examination of this problem.

The minimal case

The task is to find the *minimal case* for which edge elimination delivers a lower overall Markovitz degree than the optimal vertex elimination sequence. We will use a constructive approach for setting up this scenario step by step.

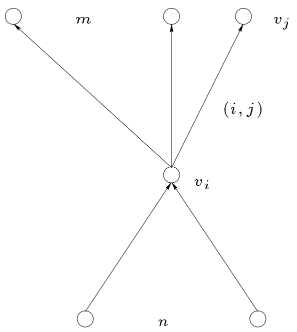

Figure 2.6: Step 2

1. We start with an intermediate edge e. e is either an in- or an out-edge of some intermediate vertex v_i (w.l.o.g. out-edge). If it was only incident with minimal and maximal vertices then it would be *final* (as in Section 1.2.1) and we could not eliminate it – neither forward nor backward.

2. Suppose v_j is the target of $e = (i,j)$ and it is maximal. v_i has arbitrary numbers of minimal predecessors and maximal successors. Obviously, we then have that the forward elimination of each of the out-edges of v_i takes always $|P_i| = n$ multiplications as there is no *absorption* (see Section 1.2) of in-edges of v_i

possible. Thus, there is no difference between the numbers of multiplications performed when eliminating (i,j) and then v_i and the straight elimination of v_i. We denote this fact by

$$((i,j),v_i) = (v_i).$$

More general we would have to write $((i,j)_{\mathbf{F}},v_i) = (v_i)$ in order to make clear that (i,j) is eliminated forward. Since the direction in which (i,j) has to be eliminated is uniquely defined by (i,j) being backward final we skip the subscript.

Summarizing the above we observe that the best vertex and the best edge elimination sequences yield the same minimal overall Markovitz degree if there is just one intermediate vertex in the c-graph.

3. We add a second intermediate vertex v_k. If $v_k \not\prec^* v_i$ and $v_i \not\prec^* v_k$ (no dependency between the two corresponding intermediate variables) then we can apply the result of the above argumentation exploiting the *divide and conquer principle*. We could divide the problem into two mutually independent subproblems which will both correspond to the situation that we have described in Step 2. It follows that we must have $v_k \prec^* v_i$ or $v_i \prec^* v_k$. v_k is either a direct predecessor of v_i or it is the target of (i,j). For any other setup we would need at least one additional intermediate vertex which we do not have, so far.

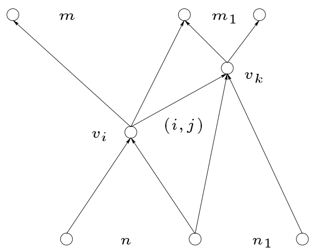

Figure 2.7: Step 4

4. Suppose $v_k = v_j$ is the target of (i,j) as shown in Figure 2.7. Neither the forward elimination of (i,j) depends on its target nor does backward elimination depend on its source. The forward elimination of (i,j) costs n and its backward elimination involves m_1 multiplications. The problem can be split into two stages:

1. The (forward or backward) elimination of (i,j).
2. The handling of two separate intermediate vertices v_i and v_k which do not depend on each other, making the divide and conquer principle applicable.

It can be checked that in either case (forward or backward elimination of (i,j)) there is no discrepancy to the pure vertex elimination mode. Therefore, the second intermediate vertex v_k cannot be the target of (i,j). It must be a predecessor of v_i.

5. As $((i,j),v_i,v_k) = (v_i,v_k)$ we need to look at $((i,j),v_k,v_i)$. Consider the situation where v_k is the only predecessor of v_i and v_i is the only successor of v_k. This is illustrated in Figure 2.8. Then we have with $|P_k| = n$ and $|S_i| = m+1$ for the operations counts of the different elimination sequences:

$$
\begin{aligned}
(v_i,v_k) &\ \hat{=}\ mn+m+n+1 && (1)\\
(v_k,v_i) &\ \hat{=}\ 2n+mn && (2)\\
((i,j),v_k,v_i) &\ \hat{=}\ 1+2n+mn && (3)
\end{aligned}
$$

(3) will never be cheaper than (2). (3) will be cheaper than (1) for $m > n$ and (1) delivers a lower operations count than (2) for $n > m + 1$. This seems to be the first situation where edge elimination could be better than pure vertex elimination. We try $m = 3$ and $n = 2$. It is easy to verify that in this case $(v_i, v_k)\hat{=}12$ and $((i,j), v_k, v_i)\hat{=}11$. However, $(v_k, v_i)\hat{=}10$ implying that (v_i, v_k) is not the optimal vertex elimination sequence which we wanted to compare with. In fact, the separate elimination of (i,j) makes sense only if the elimination of v_k before v_i delivers the optimal elimination sequence since we always have $((i,j), v_i) \equiv (v_i)$.

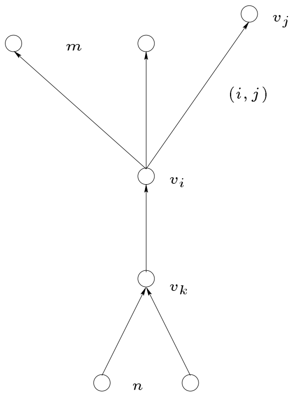

Figure 2.8: Step 5

6. Under which circumstances would there be a difference between (i,j) and the other m out-edges of v_i? (i,j) and the remaining out-edges of v_i have a common source and different targets which are all maximal vertices. So, the only thing that could make the difference is an in-edge of the target of (i,j). We call it f. It can either have v_k or any of the minimal vertices as its source. In the latter situation the local partial derivative that labels f would not be involved in any multiplication as f would be final. Therefore, we must have that v_k is the source of $f \equiv (k,j)$.

7. We have reached the first situation (see Figure 2.5) where we possibly could have

$$((i,j), v_k, v_i) < \min\{(v_k, v_i), (v_i, v_k)\}.$$

Furthermore, we must have $(v_k, v_i) < (v_i, v_k)$ giving $2n < m + 1$. We are looking for the case where $((i,j), v_k, v_i) < (v_k, v_i)$. This which implies that $1 < n$.

Getting more general

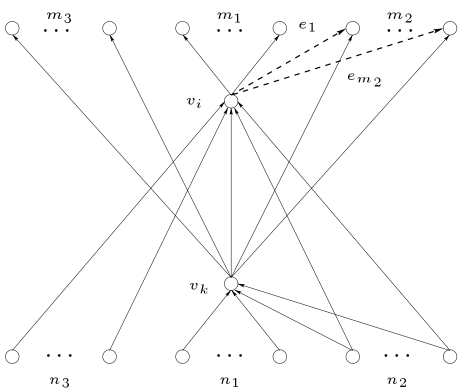

Figure 2.9: $\mathcal{D}_2$

In the previous section we have derived the minimal setup for which it is possible to observe a discrepancy between vertex and edge elimination in terms of their minimal costs. We need at least two intermediate vertices for that.

We could make the number of edges of the same kind as (i,j) variable. Say there are m_2 of them. From our construction follows that we can have the same kind of edges connecting a minimal vertex with v_k. Would that have an impact on the solution of our problem? In fact, we have discovered that the elimination of an edge e does make a difference only if we do not eliminate a vertex incident with e straight after. It follows that we have to consider either edges leading into v_k or edges emanating from v_i. The two problems are mutually exclusive. W.l.o.g. we will continue looking at out-edges of v_i.

Figure 2.9 shows the general situation for two intermediate vertices. Checking the relevant elimination orders gives us

$$\begin{aligned}
(v_i, v_k) &= (m_1 + m_2)(n_2 + n_3 + 1) + (m_1 + m_2 + m_3)(n_1 + n_2), \\
(v_k, v_i) &= (n_1 + n_2)(m_2 + m_3 + 1) + (n_1 + n_2 + n_3)(m_1 + m_2), \\
(e_1, \ldots, e_{m_1}, v_k, v_i) &= m_2(n_2 + n_3 + 1) + (n_1 + n_2)(m_2 + m_3 + 1) + m_1(n_1 + n_2 + n_3).
\end{aligned}$$

Building on the above we can formulate the following discrete maximization problem:

$$\textbf{MAXIMIZE} \quad \frac{\min\{(v_i, v_k), (v_k, v_i)\}\}}{(e_1, \ldots, e_{m_1}, v_k, v_i)} \tag{2.3}$$

subject to $n_1 \geq 1$, $m_1 \geq 1$, $n_2, n_3, m_2, m_3 \geq 0$.

We have decided not to put the constraint for *"When does edge elimination make a difference?"* into the formulation of the problem as it is implicitly in there anyway. From now on we will refer to the above maximization problem as the **(special) vertex-edge discrepancy problem**.

We observe that the objective function is monotonically decreasing with respect to both n_3 and m_3 whereas it has different peaks in the projection to n_1, n_2, m_1, and m_2. Associated with x the different variables behave like

$$\frac{a + bx}{c + dx} \quad \Rightarrow \quad \begin{cases} \text{monotonically increasing} & \text{for} \quad b > d \\ \text{monotonically decreasing} & \text{for} \quad b < d \\ \text{and} \quad \begin{cases} \text{monotonically increasing} & \text{for} \quad a > c \\ \text{monotonically decreasing} & \text{for} \quad a < c \end{cases} & \text{for} \quad b = d \end{cases}.$$

For n_3 and m_3 we have that $a = c$. If $b = d$ the behavior of the function regarding its monotonicity does not change resulting in the non-existence of any peak in the plots for both n_3 and m_3. It follows that it is sufficient to consider $n_3 = 0$ and $m_3 = 0$. This leads to a reduced version of the special vertex-edge discrepancy problem:

$$\textbf{MAXIMIZE} \quad \frac{\min\{(m_1 + m_2)(2n_2 + n_1 + 1), (n_1 + n_2)(2m_2 + m_1 + 1)\}}{m_2(n_2 + 1) + (n_1 + n_2)(m_2 + m_1 + 1)} \tag{2.4}$$

subject to $n_1 \geq 1$, $m_1 \geq 1$, $n_2, m_2 \geq 0$.

In order to solve the problem given by (2.4) it turned out to be useful to consider the continuous version obtained by regarding both the nominator and the denominator relative to $n \cdot m$ with $n = n_1 + n_2$ and $m = m_1 + m_2$. Since we deal with ratios for large values of both n_2 and m_2 the addition of one to their values will not have any impact on the solution of the maximization problem. Finally, we get the following continuous maximization problem to solve:

$$\textbf{MAXIMIZE} \quad \frac{\min\{x + 1, y + 1\}}{xy + 1}.$$

We treat the two implicit cases separately.

1. Suppose $x \geq y$. Then maximizing the objective function would result in making x small.

2. Analogous, if $x \leq y$ the maximization would lead to making y small.

From this symmetry follows that the objective function will have a maximum only if $x = y$. This leads to the simple optimization problem

$$\textbf{MAXIMIZE} \quad \frac{x+1}{x^2+1}.$$

Setting the first derivative to zero followed by the solution of the resulting quadratic equation gives us the following:

The maximal discrepancy between the vertex and the edge elimination method applied to computational graphs containing two intermediate vertices is equal to $\mathcal{D}_2$ where

$$\mathcal{D}_2 \quad = \quad \frac{1}{\sqrt{2}(2-\sqrt{2})} \quad = \quad 1,207\ldots$$

The only save statement that we can make for the general case with more than two intermediate vertices is that none of our heuristics for edge elimination ever led to an operations count which was lower than the best known from the vertex elimination strategies. So far, we have not succeeded in constructing an example which gives us a higher discrepancy than $\mathcal{D}_2$.

Conjecture:

The general vertex-edge discrepancy $\mathcal{D}_p$ – which is defined as the supremum of $c_{\mathbf{V}}$ regarded in the context of Equation (2.1) for arbitrary Jacobians J – for c-graphs containing p intermediate vertices is less or equal to 2.

2.3 Edge Elimination and State-of-the-art AD

For practical reasons most available software tools for AD build on the two basic modes – the forward and reverse modes – both of which have predictable temporal and spatial complexity. For dense Jacobians J the forward mode can compute linear combinations of columns of J with a complexity that depends on the number n of independent variables while the reverse mode computes linear combinations of rows of the Jacobian with a complexity depending on m, the number of dependent variables. We have introduced the principles of computing Jacobians without exploiting sparsity in Section 1.2.1.

The sparsity structure of the Jacobian J of $\mathbf{y} = F(\mathbf{x})$ is determined by the dependency relation between the independent variables $\mathbf{x}$ and the dependent variables $\mathbf{y}$. It can be obtained by computing the corresponding entries in the reachability matrix of the c-graph of F. Available algorithms for exploiting sparsity of Jacobians are built on finding an appropriate *seed matrix* S_1 $[S_2]$ for efficiently computing $c \ll \min\{n, m\}$ linear combinations

$$I\!R^{m \times c} \ni \quad B_1 = JS_1$$

of columns of the Jacobian in forward mode or $r \ll \min\{n, m\}$ linear combinations

$$I\!R^{n\times r} \ni \quad B_2 = S_2^T J$$

of rows of J in reverse mode. In some situations a combination of both directions may be useful.

The two main questions arising from this approach about how to find S_1 $[S_2]$ and how to reconstruct J from B_1 $[B_2]$ have been discussed extensively in the literature both in the contexts of divided differences [CPR74] and AD [CoMo84], [NeRa83]. An introduction into the proposed methods can be found in [GUG96]. Basically, we observe that in sparse forward mode the maximal number of nonzero entries $\hat{n}$ per row in J is a lower bound for the number of columns in S. Analogous, in reverse mode this lower bound is given as the maximal number of nonzeros $\hat{m}$ per column.

Edge elimination in c-graphs makes full use of the implicit sparsity of the given problem. Thus, 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. Notice that we can get this reduction even without solving the shortest path problem in the metagraph which we have described in Section 2.1. Chapters 3, 4, and 5 will be dedicated to different methods for solving the general edge elimination problem by heuristics or by putting certain restrictions on the metagraph. When presenting numerical test results we will compare them with the theoretical values for 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 one-way approach of the method by Newsam and Ramsdell (**NR**) which is

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

Let $v_i < v_j$ denote the situation where a vertex v_i is eliminated before v_j. The elimination order yielding

$$\forall w_k \in M: \qquad v_i < v_j \quad \Leftrightarrow \quad i < j$$

will be referred to as the **forward vertex elimination mode (V_F)**. Analogous,

$$\forall w_k \in M: \qquad v_i < v_j \quad \Leftrightarrow \quad i > j$$

is called **backward vertex elimination mode (V_B)**. Both **V_F** and **V_B** are considered to be equivalent to the corresponding edge elimination modes.

2.4 More about paths

The repeated application of the chain rule yields a general expression for the partial derivative c_{kl} in the scalar mode on scalar c-graphs:

$$c_{kl} \equiv \sum_{(l,\dots,k)} \prod_i c_{j_i j_{i+1}}$$

which is to be understood as the sum over all paths $(l, \ldots, k)$ connecting v_l with v_k of the chained products of the local partial derivatives labeling all edges in $(l, \ldots, k)$. In order to avoid confusion we should examine the c_{kl} more closely. Let v_l and v_k be connected by d distinct paths in CG. Then there are exactly

$$\sum_{i=1}^{d} \binom{d}{i} \quad = \quad 2^d - 1$$

different values c_{kl}^h, $(h = 1, \ldots, 2^d - 1)$ that could possibly be computed while traversing the metagraph. Consider the subgraph of CG which is induced by the vertices lying on paths connecting v_l with v_k and having v_l as its only minimal and v_k as its only maximal vertex. We could think of it as the c-graph for the scalar valued function $v_k = f_k(v_l)$ regarding all arguments to binary functions "coming from outside the c-graph of f" as constants. In this case $f_k'(v_l) = \partial v_k / \partial v_l$ is well-defined and we set $c_{kl} = f_k'(v_l)$. We then have $c_{kl} = c_{kl}^{2^d-1}$. So, we have to eliminate all edges lying on paths connecting v_l with v_k in order to calculate the unique final value for c_{kl}.

Talking about paths in the c-graph we make an important observation: Let the number of distinct paths in CG be denoted by $\mathcal{N}$ and let $\mathcal{L}$ stand for the sum of their total lengths. The length of a given path connecting an independent vertex with a dependent one is defined as the number of edges it consists of. If $\mathcal{N}$ is the number of products of the form $c_{j_1 i_1} c_{j_2 i_2} \ldots c_{j_k i_k}$ which are involved in the process of calculating the entries of the Jacobian naively one after the other then $\mathcal{L}$ is precisely the number of scalar multiplications that are performed.

The number of distinct paths $n(v_i)$ through one vertex in CG can be calculated by performing one forward or one reverse sweep. Let $n^b(v_i)$ $[n^f(v_i)]$ denote the number of different paths leading into [emanating from] a vertex $v_i \in CG$. Initializing

$$n^b(v_i) = 1 \quad \text{for} \quad i \in X \qquad \text{and} \qquad n^f(v_i) = 1 \quad \text{for} \quad i \in Y$$

we get

$$n^b(v_i) = \sum_{j \in P_i} n^b(v_j) \qquad \text{and} \qquad n^f(v_i) = \sum_{j \in S_i} n^f(v_j). \tag{2.5}$$

It follows that

$$n(v_i) = \begin{cases} n^b(v_i), & i \in Y \\ n^f(v_i), & i \in X \\ n^b(v_i) n^f(v_i), & i \in Z \end{cases}$$

and we get

$$\mathcal{N} = \sum_{j \in X} n(v_j) = \sum_{j \in Y} n(v_j). \tag{2.6}$$

Using the information about the number of different paths through a given vertex in the c-graph we can calculate the total length of all paths in CG again by either one forward or one reverse sweep. If we denote the total length of all paths leading into [emanating from] a given vertex v_i by $l^b(v_i)$ [$l^f(v_i)$] we get with

$$l^b(v_i) \quad = \quad \sum_{j \in P_i} \left[l^b(v_j) + n^b(v_j) \right] \quad = \quad n^b(v_i) + \sum_{j \in P_i} l^b(v_j)$$

and

$$l^f(v_i) \quad = \quad \sum_{j \in S_i} \left[l^f(v_j) + n^f(v_j) \right] \quad = \quad n^f(v_i) + \sum_{j \in S_i} l^f(v_j)$$

for the total length of all paths in CG:

$$\mathcal{L} = \sum_{j \in X} l(v_j) = \sum_{j \in Y} l(v_j) = \sum_{j \in X} l^f(v_j) = \sum_{j \in Y} l^b(v_j).$$

We have initialized

$$l^b(v_i) = 0 \quad \text{for} \quad i \in X \qquad \text{and} \qquad l^f(v_i) = 0 \quad \text{for} \quad i \in Y.$$

Consider the behavior of both $\mathcal{N}$ and $\mathcal{L}$ with respect to the elimination of edges in the c-graph. For the forward elimination of (i, j) we have

$$n^b(v_j) := n^b(v_j) \ - \ n^b(v_i) \ + \ \sum_{k \in P_i} n^b(v_k) \ - \ \sum_{k \in P_j \cap P_i} n^b(v_k)$$

while $n^f(v_j)$ remains unchanged. From $n^b(v_i) = \sum_{k \in P_i} n^b(v_k)$ follows

$$n^b(v_j) \ -= \sum_{k \in P_j \cap P_i} n^b(v_k)$$

which implies that the number of paths leading into v_j decreases if some path containing (i, j) coalesces with another path due to the absorption of (i, j) by an edge leading from $P_j \cap P_i$ to v_j. Otherwise, $n^b(v_j)$ is not changed by the elimination of (i, j). The forward elimination of (i, j) decreases the out-degree of v_i, thus, also reducing the number of different paths starting in v_i. Its in-degree is not effected at all giving

$$n^f(v_i) \ -= n^f(v_j) \quad \Rightarrow \quad n(v_i) \ -= n^f(v_j) \cdot n^b(v_i).$$

Let Y_j denote the set of indices such that for all $k \in Y_j$ v_k is a maximal vertex in CG reachable from v_j. From Equation (2.5) follows immediately that the number of paths leading into each of the v_k ($k \in Y_j$) depends linearly on the values of $n^b(v_l)$ for all v_l that are connected to v_k by a path. We have shown that the forward elimination of (i, j) may either result in a decrease of the number of paths leading into v_j, while not changing this value for any other vertex in CG, or this number may remain constant. An interesting consequence is that although the forward elimination of (i, j) always decreases $n^f(v_i)$ this does not necessarily lead to a decrease of the number of paths emanating from the minimal vertices in CG (see Equation (2.6)).

Regarding the effect which the elimination of an edge from CG will have on the total length of all different paths consider all paths containing (i, j) of which there are $n^b(v_i) \cdot n^f(v_j)$. Each of them is either reduced in length by one or coalesces with another one so that

$$\mathcal{L}_{new} \leq \mathcal{L}_{old} - n^b(v_i) \cdot n^f(v_j).$$

This is true for both the forward and the backward elimination of (i, j).

Finally, we can state that $\mathcal{N}$ is monotonically decreasing whereas $\mathcal{L}$ is strictly monotonically decreasing during the general edge elimination process. We may conclude that edge elimination will always come to an end. This seems to be obvious but it is certainly crucial for our approach.

Nothing changes substantially with respect to $\mathcal{N}$ and $\mathcal{L}$ when we go over to vector mode. The structure of the c-graph is invariant with respect to the different modes and even the local partial derivatives are still scalars. It follows that an upper bound for the number of products involved in the accumulation of the Jacobian by the naive component-wise calculation of its entries is still $\mathcal{N}$. However, the propagation of tangent [gradient] bundles of length $q \geq 1$ results in an upper bound for the total count of scalar multiplications which is equal to $q \cdot \mathcal{L}$.

As a direct consequence of the local partial derivatives being matrices we have a different interpretation of $\mathcal{N}$ and $\mathcal{L}$ in general vector mode. Let their meaning in a graph-theoretical context remain unchanged. $\mathcal{N}$ still denotes the number of distinct paths in the c-graph while $\mathcal{L}$ stands for the sum of the total lengths of these paths. However, the products of the local partial derivatives along the paths are now chained matrix products. It follows that we could use a dynamic programming algorithm (see Chapter 4) for minimizing the number of scalar multiplications involved in the evaluation of each single matrix chain. The corresponding maximum would again give us an upper bound for the number of scalar multiplications required for the computation of the complete Jacobian which we intent to minimize.

2.5 A Priori Reductions

The effort that has to be put into finding a method for computing the Jacobian as efficiently as possible by the solution of a shortest path problem in the corresponding metagraph grows exponentially with the number of intermediate edges [vertices]. Therefore, we must be interested in maximal a priori reductions of this number without loosing the chance to find an optimal edge [vertex] elimination sequence. In this section we will discuss some of these *a priori reduction rules*. For practical reasons we will present some heuristic extensions which deliver good results in many cases and, most importantly, can be implemented efficiently. Tests will support our approach by illustrating the often remarkable decrease in the number of intermediate edges [vertices] that can be achieved.

All we need for solving the general edge elimination problem are the edges that carry nonzero elementary partial derivatives c_{ji} as labels in the c-graph. The calculation of the c_{ji} can be performed in parallel with the computation of the function value at the current argument point during the initial forward sweep which was introduced in Section 1.2.1. It is useful to distinguish between the following stages of the process of optimizing the calculation of Jacobians by edge elimination in c-graphs:

S1: Derivation of the c-graph from the evaluation routine,
S2: Initial forward sweep to calculate the elementary partial derivatives,
S3: Solution of the general edge elimination problem.

We have assumed that all obsolete vertices $v_j \in D_1 \cup D_2$ were deleted before entering stage S3.

Definition 2.5.1 *A restriction ρ on the c-graph which leads to the extraction of a subgraph M_ρ from the metagraph M is called* **optimality preserving** *if its computational effort is* $\mathbf{OPS}\{\rho\}$ *and*

$$\mathbf{OPS}\{\rho\} + \mathbf{Cost}_\rho\{J\} = \mathbf{Cost}\{J\}.$$

The **computational effort** *of a restriction ρ is defined as the number of multiplications to be performed in order to extract M_ρ.*

Notice that ρ is not necessarily a restriction in the sense of Section 2.1. The deletion of a vertex belonging to $D_1 \cup D_2$ as well as the removal of an assignment from the c-graph could be regarded as optimality preserving as their computational efforts are zero and $\mathbf{Cost}_\rho\{J\} = \mathbf{Cost}\{J\}$. On the other hand the restriction to pure vertex elimination ($\rho = \mathbf{V}$) is certainly not optimality preserving in general although $\mathbf{OPS}\{\mathbf{V}\} = 0$ but, as shown in Section 2.2, $\mathbf{Cost}_\mathbf{V}\{J\} \geq \mathbf{Cost}\{J\}$. For a restriction to be optimality preserving is a very strong demand. In fact, we will present only one restriction on the metagraph which satisfies all demands for being optimality preserving based on the a priori elimination of all vertices having Markovitz degree one.

It is useful to define the term *optimality preserving* with respect to different subgraphs of the metagraph. A certain restriction could be optimality preserving with respect to a vertex elimination method while not satisfying the required conditions for the general edge elimination strategy. A restriction ρ corresponding to the extraction of a subgraph M_ρ from the metagraph M is considered to lead to a reduction of the complexity of the general edge elimination problem through reasonable losses in the minimal overall Markovitz degree that can possibly be achieved if

$$\mathbf{Cost}_\rho = \nu \cdot \mathbf{Cost}\{J\} \qquad \text{with} \quad \nu \in I\!R, \quad \nu \text{ small.}$$

So, for example, the restriction to a pure vertex elimination strategy could be regarded as reasonable in the above sense provided that our conjecture in Section 2.2 concerning the maximal *vertex-edge discrepancy* is right.

The minimal in-degree [out-degree] of any intermediate vertex in the c-graph is one with respect to the general edge elimination strategy. It is always possible to eliminate all in-edges of an intermediate vertex backward such that only one will be left. Analogous, this works with forward elimination for the minimal out-degree, too. If we restrict ourselves to forward [backward] elimination of edges only then, according to a result by Menger (see Section 2.6), the minimal in-degree [out-degree] is equal to the number of disjoint paths connecting the minimal vertices with its source [its target with the dependent variables]. For a pure vertex elimination method we have this result for both the minimal in-degree and the out-degree of any intermediate vertex.

We observe that the forward [backward] elimination of an edge (i, j) is optimality preserving if its source [target] is a hoisting vertex, i.e. if it has exactly one predecessor and one successor. Let therefore α denote the forward elimination of (i, j), which is equivalent to the elimination of its source v_i. We have that $\mathbf{OPS}\{\alpha\} = 1$

since v_i is a hoisting vertex. Obviously, there is no way to eliminate an intermediate edge [vertex] at a lower cost. Suppose that keeping (i, j) will result in the possibility to find a shorter path in M, i.e.

$$\mathbf{OPS}\{\alpha\} + \mathbf{Cost}_\alpha\{J\} > \mathbf{Cost}\{J\}.$$

The in-degree [out-degree] of vertices in the c-graph can be decreased if and only if any of their in-edges [out-edges] are involved in the absorption of other edges. In our situation this would imply that there is at least one other path connecting v_i with v_j apart from (i, j). This would lead to a contradiction to v_i being a hoisting vertex. The forward elimination of (i, j) is equivalent to the backward elimination of the only in-edge (k, i) of v_i which is optimality preserving by the similar argument.

With the help of the knowledge of the bounds for vertex degrees it will be possible to define a priori reductions that are optimality preserving with respect to different subgraphs of the metagraph. In Section 2.6 we will show that both the problems of finding a lower and an upper bound for the vertex degrees are NP-hard. Heuristics turned out to deliver very promising results as we will see in the next section.

2.5.1 Heuristics for a priori reductions

Forward [backward] invariant edges

Definition 2.5.2 *We call an edge* (i, j) **forward [backward] invariant** *if it can be eliminated at a cost of one multiplication and one addition, i.e. if the single predecessor* v_p *[successor* v_s*] of* v_i *[*v_j*] is connected with* v_j *[*v_i*] by an edge* (p, j) *[*(i, s)*].*

To make sure that the forward [backward] elimination of an invariant edge (i, j) is optimality preserving we would have to check whether the out-degree of v_i [in-degree of v_j] is minimal. For the general edge elimination approach this would again lead to the necessity for v_j [v_i] to be a hoisting vertex. However, numerous test results support the usefulness of this heuristic approach as it usually leads to a significant decrease of the number of edges in the c-graph. Moreover, it is a fast and easy to implement way for decreasing the complexity of the combinatorial optimization problem leading in most cases to only reasonable losses in the overall cost.

The forward [backward] elimination of forward [backward] invariant edges may lead to its source [target] becoming a hoisting vertex. Therefore, this heuristic should be run in parallel with the elimination of all hoisting vertices resulting in an even larger reduction of the size of the c-graph.

We have extended the above idea to what we call *forward [backward] fill-in invariant edges*. In this case we do not require the cost of eliminating an edge to be one multiplication. However, every multiplication must be followed by an addition, thus, making optimal use of absorption as all newly generated edges are immediately absorbed by an already existing edge.

Finally, we have implemented an even weaker heuristic based on the exploitation of absorption. Whenever an edge (i, j) is part of a "triangle" (i.e. if there exists a direct predecessor v_p of v_i which is connected with v_j by an edge (p, j) [if there exists a direct successor v_s of v_j such that there is an edge (i, s)]) we eliminate it forward [backward].

Additive parts

Consider a function of the form $y = f(x, q)$ with $f : I\!R^n \to I\!R$ and

$$f_b(x,q) = y = \psi\left(\sum_{j=0}^{q-1} \varphi_j\left(\frac{\sum_{i=0}^{n-1} x_i}{j+1}\right)\right)$$

which we will refer to as the so-called *belt function* indicating that its behavior with respect to the cost of the vertex elimination approach can be improved significantly by "tightening its belt". This will become clear during the following discussion. Naively, the fact that we have additive parts in the c-graph could be a reason for eliminating them a priori because we could avoid the trivial multiplications by [minus] one which, obviously, are the corresponding partial derivatives labeling the edges that lead into the vertices representing the addition [subtraction]. The original c-graph of the problem is shown in Figure 2.10 on the left. We will look at two different reduced c-graphs (CG in the middle of Figure 2.10 and $\hat{CG}$ on the right) obtained by performing certain a priori reductions. Suppose that the vertices in $\hat{CG}$ are labeled as follows:

$$v_{i-n+1} = x_i, \quad i = 0, \ldots, n-1, \quad v_1 = \sum_{i=0}^{n-1} x_i,$$

$$v_j = \varphi_{j-2}(v_1), \quad j = 2, \ldots, q+1, \quad v_{q+2} = \sum_{i=2}^{q+1} v_i.$$

Let CG be labeled in a similar way, i.e. just with v_1 is missing. As usual, the $c_{ji} = \partial v_j / \partial v_i$ denote the local sensitivity of v_j with respect to v_i for $j = 1, \ldots, q+2$ and $i = 1-n, \ldots, q+1$. Applying the backward vertex elimination mode (**V_B**) to both CG and $\hat{CG}$ we observe a very interesting behavior of the cost of accumulating the gradient depending on the parameters n and q. We denote the cost of running **V_B** on CG [$\hat{CG}$] by $\mathbf{OPS}_{\mathbf{V\_B}}(CG)$ [$\mathbf{OPS}_{\mathbf{V\_B}}(\hat{CG})$].

Case 1 (CG):

$$\frac{\partial y}{\partial x_{i+n-1}} = \frac{\partial v_{q+2}}{\partial v_i} = \sum_{j=1}^{q} c_{ji} \cdot c_{q+2,j} \qquad \text{for } i = 0, \ldots, n-1.$$

The cost of running this elimination sequence is

$$\mathbf{OPS}_{\mathbf{V\_B}}(CG) = q(n+1).$$

Case 2 ($\hat{CG}$):

$$c_{q+2,1} = \sum_{j=2}^{q+1} c_{q+2,j} \cdot c_{j,1},$$

$$\frac{\partial y}{\partial x_{i+n-1}} = \frac{\partial v_{q+2}}{\partial v_i} = 1.0 \cdot c_{q+2,1} \qquad \text{for } i = 0, \ldots, n-1.$$

In this case the multiplications with 1.0 are actually performed and we end up with a cost of

$$\mathbf{OPS}_{\mathbf{V\_B}}(\hat{CG}) = 2q + n.$$

We have assumed that the calculation of the c_{ji} involves the multiplication with the constant factor $1/(j+1)$ which we have also omitted in the representation of the c-graph in Figure 2.10. Obviously, we have that

$$\lim_{q\to\infty}\left\{\frac{\mathbf{OPS}_{\mathbf{V\_B}}(CG)}{\mathbf{OPS}_{\mathbf{V\_B}}(\hat{CG})}\right\} = \frac{1}{2}(n+1),$$

$$\text{and} \quad \lim_{n\to\infty}\left\{\frac{\mathbf{OPS}_{\mathbf{V\_B}}(CG)}{\mathbf{OPS}_{\mathbf{V\_B}}(\hat{CG})}\right\} = q$$

which implies that depending on q and n **V_B** on CG can become arbitrarily worse than the same vertex elimination strategy applied to $\hat{CG}$. The reason for this is the repeated recalculation of *common subexpressions* as a result of the a priori removal of all additive parts from the c-graph. In terms of accumulating Jacobians by edge elimination we can associate this effect with the generation of fill-in. Based on this observation we have implemented a heuristic version of the handling of additive parts which deletes a vertex representing an addition or subtraction only if the generated fill-in is reasonable low. Specifically, given a vertex v_i which represents an additive operation we check whether at the current stage w_k in the metagraph we have

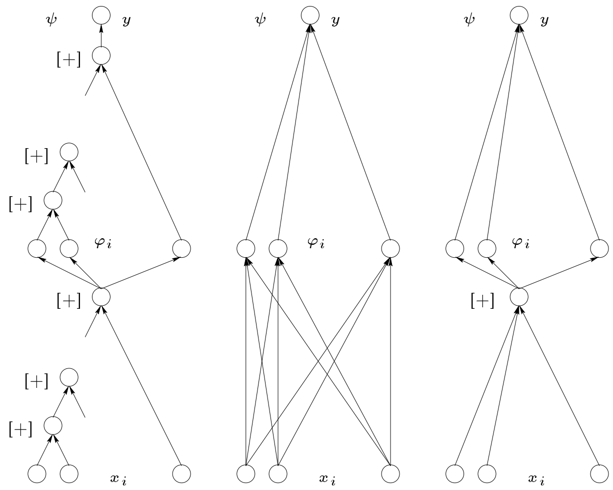

Figure 2.10: Belt Function

$$|P_i|_k \cdot |S_i|_k \le \nu(|P_i|_k + |S_i|_k)$$

where $\nu \in I\!R$ is a small constant which we have set to two in the current implementation. Our test results show that this approach is very effective.

Sub-minimal and sub-maximal vertices

We denote the minimal [maximal] in-degree of a vertex v_i with respect to all stages in the metagraph by $\underline{|P_i|}$ [$\overline{|P_i|}$] and we use similar notations for the out-degree.

Definition 2.5.3 *Let $CG = (V, E)$ be a c-graph.*

1. *A vertex $v_i \in V$ is called* **sub-minimal** *if $\underline{|P_i|} = \overline{|P_i|}$.*

2. *A vertex $v_i \in V$ is called* **sub-maximal** *if $\underline{|S_i|} = \overline{|S_i|}$.*

In the general metagraph M a vertex can only be sub-minimal [sub-maximal] if it is a hoisting vertex. This is no longer true for certain restrictions on the metagraph leading to subgraphs like for instance $M_{\mathbf{V}}$.

Building on the concept of sub-minimality [sub-maximality] we have implemented more heuristics for eliminating vertices in the c-graph a priori. We have concentrated on regarding vertices that have only one minimal [maximal] predecessor [successor]. As a first heuristic we successively eliminate all these vertices regardless of their out-degrees [in-degrees]. Secondly, only those vertices with an out-degree [in-degree] which is less or equal to two are eliminated.

2.5.2 Case study

We will discuss three `MINPACK` [ACM91] test problems.

Chebyshev Quadrature Problem (n=50, m=51): Let us have a look at two different approaches to the a priori reduction of the c-graph for $n = 50$ independent and $m = 51$ dependent variables. It consists of 7826 vertices and 15225 edges. In the first case we have removed all additive parts leading to a c-graph containing 2726 vertices and 68925 edges. Notice the large fill-in that led to an increase of the number of edges by a factor of more than 4. We have applied both the forward and the backward vertex elimination modes to this c-graph resulting of operations counts of 70100 and 622500 respectively. In a second test we have applied the degree-bound-based approach to the a priori elimination of additive parts. This time the resulting c-graph contained 3076 vertices and 22775 edges which is a reduction of the generated fill-in by a factor of 3. The operations counts for both **V_F** (23950) and **V_B** (325850) could be decreased remarkably.

By keeping 350 vertices and thus by performing the corresponding multiplications with one we avoid the large artificial fill-in and we exploit the effect of absorption leading to a significant decrease of the in-degrees in the forward and of the out-degrees in backward vertex elimination modes.

Coating Thickness Standardization Problem (n=134, m=252): This problem is an example for the a priori reductions leading to the elimination of all intermediate vertices. Starting with a c-graph that contains 1772 vertices and 2520 edges the degree-bound elimination of all sub-maximal vertices having a maximal vertex as their only predecessor leads to the subgraph of the $K_{n,m}$ we are striving for. This is verified by the identical result obtained by running the backward vertex elimination mode.

Gaussian data fitting problem (n=11, m=65): Our third example is supposed to illustrate the efficiency of eliminating all invariant edges and hoisting vertices. For the given problem it results in a decrease of the number of intermediate vertices from 1051 to 531. In Chapters 3, 4, and 5 we will present several approaches to the solution of the general edge elimination problem in c-graphs. The run-times of these algorithms are decreased significantly by the reduction.

2.6 Bounds for Vertex Degrees

The solution of the general edge elimination problem results in an elimination sequence of all intermediate edges and vertices in the c-graph which can be applied using a minimal number of multiplications. The vertex degrees are the variables in this optimization process. Obviously, the degree of a given vertex varies within a certain interval taking selected discrete integer values from it. There will be both a maximum and a minimum for the in-degree [out-degree] that a vertex can have at any stage of the metagraph. The knowledge of these values would be advantageous in many situations.

Definition 2.6.1 *Let $CG = (V, E)$ be a c-graph with $V = X \cup Z \cup Y$. A* **predecessor set** *of a vertex $v_s \in Y \cup Z$ is a set of vertices*

$$\mathbf{ps}(v_s) = \{v_1, \dots, v_k\}, \quad v_i \in Z \cup X,$$

for which

$$\exists w_j \in M : \quad \forall v_i \in \mathbf{ps}(v_s) \;\; \exists (i,s) \in E_j.$$

$\mathbf{cps}(v_s)$ *is called a* **complete predecessor set** *of v_s if it is a predecessor set of v_s and if it additionally satisfies*

$$\nexists v_l \in X \;:\; \exists (l, \dots, s) \;:\; \forall v_j \in \mathbf{cps}(v_s) : v_j \notin (l, \dots, s).$$

Analogous, a **successor set** *of a vertex $v_s \in X \cup Z$ is a set of vertices*

$$\mathbf{ss}(v_s) = \{v_1, \dots, v_k\}, \quad v_i \in Z \cup Y,$$

with

$$\exists w_j \in M : \quad \forall v_i \in \mathbf{ss}(v_s) \;\; \exists (s,i) \in E_j$$

and it becomes a **complete successor set** *of v_s, denoted by $\mathbf{css}(v_s)$, if*

$$\nexists v_l \in Y \;:\; \exists (s, \dots, l) \;:\; \forall v_j \in \mathbf{css}(v_s) : v_j \notin (s, \dots, l).$$

We are going to look for minimal and maximal complete predecessor [successor] sets. First of all we observe that it is sufficient to ask for bounds for the in-degree [out-degree] of vertices. Thus, we will also get the corresponding information for the edges in the c-graph CG via their sources or targets. Using a general edge elimination approach it is always possible to reduce the number of in-edges [out-edges] of any vertex to one. This is done by, for instance, the forward elimination of all but one out-edges of a given intermediate vertex v_i making $|S_i| = 1$. The problem of finding lower bounds for both the in- and the out-degrees of intermediate vertices [edges] in CG becomes more interesting for certain restricted elimination methods. If we do not allow forward elimination of edges it certainly will be impossible to reduce the out-degree of all vertices in CG to one. Analogous, restricting ourselves to the forward elimination of edges exclusively we lose the possibility to make the in-degrees of all vertices equal to one by successively eliminating all their in-edges backward except for one. Similar observations can be made for the pure vertex elimination strategy. Since vertex elimination will play an important role as a restriction on the metagraph in our attempt to optimize the application of the chain rule on c-graphs we will present the following discussion in a vertex elimination context. However, the obtained results will be true for certain other restrictions such as *forward edge elimination only* or *backward edge elimination only*, too. The statements about upper bounds for vertex degrees in c-graphs can be regarded in the general edge elimination context.

2.6.1 Minimum

We could conjecture that minimal and maximal edge cuts in the c-graph regarded as a network with edge capacities set to 1 could solve our problem. However, it can easily be verified that these cuts are not equivalent to the bounds that we are looking for. We have to make clear what the minimal in-degree of a given vertex actually is. Recapitulating the definition of a separating set from Section 1.1.2 we observe the following:

Theorem 2.6.1 *Let A be a set of vertices in the c-graph CG.*

1. *A is a minimal complete predecessor set of a vertex v_s if and only if it is a minimal $[X|v_s]$-separating set in CG.*

2. *A is a minimal complete successor set of a vertex v_s if and only if it is a minimal $[v_s|Y]$-separating set in CG.*

Proof:

We will prove only the first part of the theorem. The second argument is analogous.

1. (A is a complete predecessor set of $v_s \Rightarrow A$ is an $[X|v_s]$-separating set)

   This follows directly from the definition of the complete predecessor set. It says that all paths connecting a vertex from X with v_s pass through $A = \mathbf{cps}(v_s)$ which is exactly what we demand for A to be an $[X|v_s]$-separating set in CG.

2. (A is a minimal $[X|v_s]$-separating set $\Rightarrow A$ is a complete predecessor set of v_s)

   Suppose that A contains a vertex v_p such that there is no path connecting v_p with v_s. Then $A \setminus \{v_p\}$ would also be an $[X|v_s]$-separating set which gives a contradiction to the minimality of A. Therefore, we have that A consists of vertices for which there is a path connecting them with v_s exclusively. It follows that A is a predecessor set of v_s. Since A is an $[X|v_s]$-separating set we have that all paths from X to v_s pass through A which makes it a complete predecessor set of v_s.

3. (A is a minimal $[X|v_s]$-separating set $\Rightarrow A$ is a minimal complete predecessor set of v_s)

   Suppose A is a minimal $[X|v_s]$-separating set but is not minimal as a complete predecessor set of v_s. With 2. we have that A is a complete predecessor set of

v_s. Let $v_p \in A$ be such that its removal from A does not change A's property to be a complete predecessor set of v_s. However, as A is minimal as an $[X|v_s]$-separating set it cannot remain an $[X|v_s]$-separating set after the removal of v_p which gives a contradiction to 1.

4. (A is a minimal complete predecessor set of $v_s \Rightarrow A$ is a minimal $[X|v_s]$-separating set)

   This is a direct consequence of 1. and 3. ◄

One of the most important theorems which we will refer to is Menger's theorem about the maximum number of disjoint paths in a graph. The proof can be found in numerous books on graph theory, e.g. see [Har69].

Theorem 2.6.2 *The maximum number of (vertex-)disjoint paths connecting two disjoint vertex sets V and W of a graph G is equal to the minimum number of vertices in a $[V|W]$-separating set.*

The following theorem can be regarded as a direct consequence of Menger's theorem. We have shown that in a c-graph $CG = (V, E)$ a set A of vertices forms a minimal complete predecessor set of a vertex v_s if and only if it is a minimal $[X|v_s]$-separating set in CG. Menger's theorem says that the minimal number of vertices in an $[X|v_s]$-separating set is equal to the maximal number of (vertex-) disjoint paths connecting the minimal vertices with v_s in CG. Summarizing the results from Theorems 2.6.1 and 2.6.2 we get the following fundamental result:

Theorem 2.6.3 *The minimal in-degree of a vertex $v_j \in Z \cup Y$ in a c-graph is equal to the number of disjoint paths connecting the minimal vertices with v_j.*

The proof for this has already been given by the two previous results. ◄

2.6.2 Maximum

Consider the problem of finding a maximal complete predecessor [successor] set of a given vertex in the c-graph. We will start with proving two fundamental results which will lead directly to the solution.

Theorem 2.6.4 *Let $CG = (V, E)$ be a c-graph and $R_j \subseteq V$ the set of vertices that are reachable from a vertex v_j in the converse c-graph CG'. There exists a subtree $T_j \subseteq CG'$ induced by a subset of R_j with a set of leaves $L_j = \{v_{i_1}, \ldots, v_{i_k}\}$ if and only if there exists a stage w_i in the metagraph where $\{(i_1, j), \ldots, (i_k, j)\} \subseteq E_i$.*

Proof:

We choose a constructive approach to the proof.

($\Rightarrow$) Let $T_j \subseteq CG'(R_j)$ where $CG'(R_j)$ denotes the subgraph of CG' which is induced by the vertices in R_j. The elimination of all vertices representing branches in T_j leads to a stage w_i of the metagraph at which $\{(i_1, j), \ldots, (i_k, j)\} \subseteq E_i$.

($\Leftarrow$) Suppose that there is such a stage w_i in the metagraph. The reversal of the vertex elimination procedure without allowing to occur any semi-cycles delivers the subtree T_j. For all vertices in L_j choose a path from v_j to v_j in CG. Every time we meet an already existing vertex we merge the two paths. ◄

We call a spanning tree with source $v_j \in CG'(R_j)$ which has the maximal number of leaves compared to all other spanning trees **maximum width spanning v_j-arborescence**.

Theorem 2.6.5 *Let $\bar{A}_j \subseteq CG'(R_j)$ be a maximum width spanning v_j-arborescence and $T_j \subseteq CG'(R_j)$ a v_j-subtree with the maximum number of leaves. Then the number of leaves in $\bar{A}_j$ is equal to the number of leaves in T_j.*

Proof:

Obviously, as T_j has the maximum number of leaves this number cannot be less than the number of leaves in any maximum width spanning v_j-arborescence. Suppose it is greater. We will show that for every maximum width v_j-subtree $T_j \subseteq CG'(R_j)$ there is a spanning v_j-arborescence with the same number of leaves as in T_j.

The leaves of T_j form an $[L|v_j]$-separating set where L is the set of vertices in R_j which do not belong to T_j, as otherwise T_j could not be maximal with respect to the number of its leaves. Select one $v_k \in L$ such that there exists an edge (k,t) for some $v_t \in T_j$. v_t is a leaf as otherwise T_j could not be maximal. Therefore v_k and (k,t) could be added to T_j and we would get a new tree with the same number of leaves. Recursively the same argument holds for all $v_l \in L$. So, we can construct a spanning v_j-arborescence $\bar{A}_j$ out of T_j having the same number of leaves. Since T_j is maximal so is $\bar{A}_j$. ◄

The direct consequence from Theorems 2.6.4 and 2.6.5 is the following theorem which answers the question of how to compute the cardinality of a maximal complete predecessor set.

Theorem 2.6.6 *The maximal number of predecessors of a vertex v_j in a c-graph CG is equal to the number of leaves in a maximum width spanning v_j-arborescence in CG' that contains all vertices in G which are sources of a path leading to v_j.*

The proof follows from the two previous results. ◄

The general *Maximum Leaf Spanning Tree* problem can be formulated as follows: Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a spanning tree for G in which K or more vertices have degree 1? The bad news is that this problem is shown to be NP-complete. Garey and Johnson did it in an unpublished paper by a polynomial transformation from the DOMINATING SET problem.
The maximum width spanning arborescence problem can be formulated as a boolean programming problem. Consider the adjacency matrix of the subgraph H_i induced by the set R_i of all vertices in CG which are reachable from v_i. Obviously, as H_i is a graph in our sense the adjacency matrix $A_i = A(H_i)$ is a square lower triangular boolean matrix. Our objective is to determine a spanning v_i-arborescence

with maximum cardinality of the set of its leaves giving us the maximum number of predecessors that v_i can possibly have during the elimination process. Since the number of nonzero entries in the j-th column of A_i is equal to the number of predecessors of v_j we want to convert A_i in a way that every column contains just one nonzero entry and with the number of rows not containing any nonzero entry maximized. This last number is then equal to the number of leaves in the maximum width spanning v_i-arborescence which is exactly the information we would like to obtain. Obviously, v_i will always have to be one of the branches.

With $\tilde{A} = A \cup I_0$, where I_0 is defined in Section 1.2.3, and $\mathbf{z} = (z_0, \dots, z_{q-1})$, $\mathbf{z} \in \{0,1\}^q$ such that

$$z_j = \begin{cases} 0 & v_j \text{ is a leaf} \\ 1 & v_j \text{ is a branch} \end{cases}$$

we get the following boolean optimization problem to solve:

$$\textbf{MINIMIZE} \qquad w = \mathbf{e}^T\mathbf{z} \quad \textbf{SUBJECT TO} \quad \tilde{A}\mathbf{z} \geq \mathbf{e}, \;\; \mathbf{z} \in \{0,1\}^q \;\; \mathbf{e} \equiv \{1\}^q.$$

As a result we get a boolean solution vector $\mathbf{z}^*$, the zero entries of which stand for the leaves in (one of) the maximum width spanning v_i-arborescence. We finally get

$$\overline{|P_i|} \;\; = \;\; q - \sum_{j=0}^{q-1} z_j^*.$$

Chapter 3

Heuristics

Building on the possibility to apply the chain rule in arbitrary orders which results in different operations counts for the calculation of the Jacobian matrix we have described the computationally hard combinatorial optimization problem of determining edge elimination sequences that minimize this number of arithmetic operations. Heuristics for finding the required elimination orders play an important role in the search for a solution of the general edge elimination problem.

3.1 Heuristics for Vertex Elimination

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 Markovitz degrees at each stage of the metagraph. At a particular stage $w_k \in M_{\mathbf{V}}$ we eliminate the vertex with the lowest Markovitz degree among all intermediate vertices. This approach is known as the **Lowest-Markovitz-Degree-First** strategy and it represents a robust and easy to implement way of calculating elimination sequences. A more formal description of this heuristic is given below. It can be combined with different tie-break criterions which could improve its performance.

Lowest-Markovitz-Degree-First heuristic (V_LM):

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

At each stage in the metagraph we have to compute the number of predecessors and successors of every vertex which results in a complexity of $O(|V|^2)$ for **V_LM**. As a well-known heuristic for the minimization of the generated fill-in during the solution of sparse systems of linear equations the Markovitz 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.

This is motivation enough for us to develop different classes of new heuristics most of which result in nearly optimal elimination sequences in many cases. With every basic idea we can combine numerous versions of heuristics belonging to the same class and there is always enough room for experiments and improvements. Here, we will restrict ourselves to the introduction of the framework by presenting one representative of each class.

3.1.1 Dependency degree based heuristics

Definition 3.1.1 *Let $CG = (V, E)$ be a c-graph.*

1. *A vertex $v_j \in Y \cup Z$ has the* **input-dependency degree** $\mathtt{id}_j = k$ *if there are paths connecting a maximum of k minimal vertices with v_j.*

2. *A vertex $v_j \in X \cup Z$ has the* **output-dependency degree** $\mathtt{od}_j = k$ *if a maximum of k maximal vertices are reachable from v_j.*

3. *For vertices $v_j \in Z$ we define the* **dependency degree** *to be* $\mathtt{dd}_j \equiv \mathtt{id}_j \cdot \mathtt{od}_j$.

The dependency degree $\mathtt{dd}_j$ of an intermediate vertex v_j is equal to the Markovitz degree at which v_j would be eliminated if it was the last in the elimination sequence. The naive approach to how cheaply a particular vertex v_j can be eliminated would lead to the earlier discussed Lowest-Markovitz-Degree-First heuristic. A different method could be to evaluate the actual Markovitz degree of v_j relative to some value that is invariant with respect to different elimination sequences. The case in which v_j is eliminated at the highest ($\overline{|P_j|} \cdot \overline{|S_j|}$) [lowest ($\underline{|P_j|} \cdot \underline{|S_j|}$)] cost is one possibility. As the problem of finding $\overline{|P_j|}$ and $\overline{|S_j|}$ [$\underline{|P_j|}$ and $\underline{|S_j|}$] is in general expensive (see Section 2.6) we help ourselves by exploiting the fact that the dependency degree of a vertex is invariant with respect to the pure vertex elimination strategy and that $\mathtt{dd}_j$ is close to the maximal possible Markovitz degree of v_j in many cases. This property makes the dependency degree ideal for the implementation of new heuristics. Once we have tabulated it for every vertex in the c-graph, which can be done at a cost of $O(|V|(|V| + |E|))$ [MeNä96], we can look it up at every stage $w_k \in M_{\mathbf{V}}$. The computation of the reachability matrix dominates the entire calculation. The dependency degree of a vertex must not be regarded as constant in the context of the general edge elimination method. For a vertex v_j reachable from at least two minimal vertices it is always possible to reduce its input-dependency degree by one (and up to one). Suppose that we want to eliminate the dependency of v_j from $v_i \in X$. We simply have to eliminate all edges belonging to paths which connect v_i with v_j forward followed by the backward elimination of the resulting edge (i, j). One possible instance of a dependency degree based heuristic could be the

Lowest-Relative-Markowitz-Degree-First heuristic (V_LR):

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

The **V_LR** strategy is slightly more expensive than **V_LM** due to the single calculation of $\mathtt{id}_j$ and $\mathtt{od}_j$ for all intermediate vertices. In many cases it delivers better elimination sequences which more than compensate the additional effort. It could be worth thinking about some tie-break criterion that could possibly even improve the performance of the generated elimination sequences. Extensive tests showed that in general **V_LR** does not give the optimal elimination sequence. But still it is a fast and practicable way of finding nearly optimal solutions in most cases. It turned out to be the most consistent strategy in comparison with **V_F**, **V_B**, and **V_LM**.

Example

We will continue with the example from Section 1.2.5. The application of the a priori reduction rules to the complete c-graph CG_F^c displayed in Figure 1.8 leads to its reduced version which is shown in Figure 3.1 and which contains as little intermediate vertices as possible.

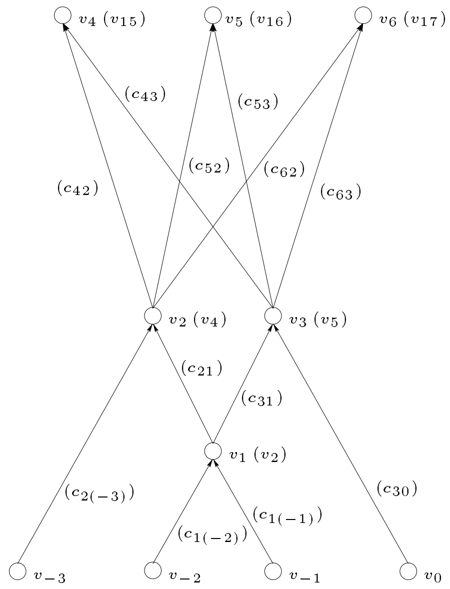

Figure 3.1: CG

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 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.

3.1.2 Fill-in based heuristics

As a result of the elimination rule the elimination of a vertex v_i at a stage w_k in the metagraph causes the insertion of

$$f_i^k = |P_i|_k \cdot |S_i|_k - (|P_i|_k + |S_i|_k) - \zeta_i^k \tag{3.1}$$

new edges into the c-graph where ζ_i^k is the number of edges connecting predecessors of v_i with its successors. 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 heuristic (V_LF)

$$w_k \in M_{\mathbf{V}} : \quad v_i < v_j \quad \Leftrightarrow \quad f_i^k \;\leq\; f_j^k .$$

The relatively high time complexity ($O(|V|^3)$) of this heuristic is caused by the repeated computation of ζ_i^k. Therefore, we have also implemented a reduced version where we simply consider $|P_i|_k \cdot |S_i|_k - (|P_i|_k + |S_i|_k)$ which leads to a $O(|V|^2)$ algorithm.

3.1.3 $\mathcal{L}$-based heuristics

Let the total length $\mathcal{L}$ of all distinct paths in the c-graph CG be defined as the sum of the numbers of edges contained within each of them. In Section 2.4 we have observed that in scalar mode on scalar c-graphs this number represents an upper bound for the number of scalar multiplications involved in the accumulation of the complete Jacobian. $\mathcal{L}$ is strictly monotonically decreasing along every path in the metagraph. Thus, we have concluded that edge elimination in c-graphs will always terminate in form of a subgraph of the complete bipartite graph with a bipartition corresponding to the independent and the dependent variables.

We have developed two heuristics that exploit the above observations and we will present them both. Notice that $\mathcal{L}$ is not a static quantity. It depends on the stage in the metagraph of a given elimination problem. At a stage w_k in the vertex metagraph $M_{\mathbf{V}}$ we denote the total length of all paths in CG_k by $\mathcal{L}_k$. For any vertex path in $M_{\mathbf{V}}$ $(w_{k_1}, \ldots, w_{k_p})$ (each of them has length $p \equiv |Z|$) we have

$$\mathcal{L}_{k_1} > \ldots > \mathcal{L}_{k_p}$$

where $\mathcal{L}_{k_p}$ is equal to the number of nonzero entries in the Jacobian. $\mathcal{L}_{k_p}$ can also be regarded as the sum of the input-dependency degrees of all maximal vertices which is precisely the same as the sum of the output-dependency degrees of all minimal vertices.

The idea behind our first $\mathcal{L}$-based heuristic is to find a path in $M_{\mathbf{V}}$ such that the difference $\mathcal{L}_{k_i} - \mathcal{L}_{k_{i+1}}$ is maximized by eliminating a vertex which causes the maximum decrease of the upper bound for the number of scalar multiplications required for accumulating the Jacobian.

Maximum-Total-Path-Length-Reduction-First heuristic (V_RPL):

Suppose that starting at a stage w_k in the metagraph the elimination of a vertex $v_i \in CG_k$ leads to stage $w_{k_i} \in M_{\mathbf{V}}$ and the elimination of $v_j \in CG_k$ is equivalent to move to stage $w_{k_j} \in M_{\mathbf{V}}$. Then

$$w_k \in M_{\mathbf{V}}: \quad v_i < v_j \quad \Leftrightarrow \quad \mathcal{L}_k - \mathcal{L}_{k_i} \;\geq\; \mathcal{L}_k - \mathcal{L}_{k_j}$$

which is the same as

$$w_k \in M_{\mathbf{V}}: \quad v_i < v_j \quad \Leftrightarrow \quad \mathcal{L}_{k_i} \;\leq\; \mathcal{L}_{k_j}$$

and it can be implemented at a cost of $O(|V|^3)$. The above approach does not take into account the actual Markovitz degrees of the vertices which are the costs of eliminating them from the c-graph. Since our objective is to minimize the overall Markovitz degree by finding a shortest path in the vertex metagraph it certainly would make sense to include this information into our heuristic. Therefore, we have implemented a second version where we regard the new total path length caused by the elimination of a vertex $v_i \in CG_k$ weighted with its actual Markovitz degree $|P_i|_k \cdot |S_i|_k$ at the current stage $w_k \in M_{\mathbf{V}}$.

Minimum-Relative-Total-Path-Length-First heuristic (V_RLR):

$$w_k \in M_{\mathbf{V}}: \quad v_i < v_j \quad \Leftrightarrow \quad (|P_i|_k \cdot |S_i|_k) \cdot \mathcal{L}_{k_i} \;\leq\; (|P_j|_k \cdot |S_j|_k) \cdot \mathcal{L}_{k_j}.$$

The implementation of the $\mathcal{L}$-based heuristics is rather expensive as at each stage w_k in the metagraph and for all vertices in CG_k one has to perform one sweep through the c-graph in order to compute the new total path length. Thus, we end up with an algorithm of order $O(|V|^3)$ for **V_RLR**.

3.1.4 $\mathcal{N}$-based heuristics

We denote the number of distinct paths passing through a vertex $v_i \in CG$ by $n(v_i)$, as in Section 2.4. The fact that this number also depends on the stage $w_k \in M_{\mathbf{V}}$ will be denoted by $n_k(v_i)$. Building on this knowledge we have implemented a heuristic where we consider the actual Markovitz degree of $v_i \in CG$ relative to the number of paths leading through it. The idea is that if $|P_i|_k \cdot |S_i|_k$ is small at the current stage w_k in the metagraph and if there are many paths leading through v_i then the Markovitz degree of v_i is likely to be increased at a later stage of the elimination process. Similarly to the *Lowest-Relative-Markovitz-Degree-First* heuristic this approach can be regarded as a *Lowest-Markovitz-Degree-First* strategy extended by some global information represented by the number of paths leading through a vertex.

Extended-Lowest-Markovitz-Degree-First heuristic (V_ELM):

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

Since we have to compute both the number of paths through a vertex and its Markovitz degree at each stage in $M_{\mathbf{V}}$ we end up with an algorithm the complexity of which is $O(|V|^3)$.

3.2 Heuristics for Edge Elimination

The ideas behind the heuristics for finding nearly optimal edge elimination orders are similar to the ones exploited in the vertex case. Undoubtedly, there are equivalents for both the forward and backward elimination modes provided that an appropriate re-numbering scheme which will be necessary due to the generated fill-in is used.

Fill-in based heuristics

Remember that the fill-in caused 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. The fill-in generated by the forward elimination of an edge (i, j) is given by

$$\begin{array}{rll} & \mathtt{ffi}\{(i,j)\} = |S_j| - \overline{\kappa}_{ij} - 1 & \text{if } |P_j| > 1 \\ \text{or} & \mathtt{ffi}\{(i,j)\} = -(\overline{\kappa}_{ij} + 1) & \text{if } |P_j| = 1 \end{array}$$

where $\overline{\kappa}_{ij}$ denotes the number of common successors of v_i and v_j. Analogous, the fill-in generated by the backward elimination of the same edge is given by

$$\begin{array}{llr} & \texttt{bfi}\{(i,j)\} = |P_i| - \underline{\kappa}_{ij} - 1 & \text{if } |S_i| > 1 \\ \text{or} & \texttt{bfi}\{(i,j)\} = -(\underline{\kappa}_{ij} + 1) & \text{if } |S_i| = 1 \end{array}$$

where $\underline{\kappa}_{ij}$ denotes the number of common predecessors of v_i and v_j. We have implemented a procedure which calculates the values for $\texttt{ffi}\{(i,j)\}$ and $\texttt{bfi}\{(i,j)\}$. Depending on which of them is lower we eliminate (i,j) either forward or backward:

$$w_k \in M : \quad (i_1,j_1) <_{[f,b]} (i_2,j_2)$$

$$\Leftrightarrow$$

$$\min\{\texttt{ffi}\{(i_1,j_1)\}, \texttt{bfi}\{(i_1,j_1)\}\} \leq \min\{\texttt{ffi}\{(i_2,j_2)\}, \texttt{bfi}\{(i_2,j_2)\}\}.$$

We will refer to this heuristic as the **Lowest-Fill-in-First** edge elimination heuristic (**E_LF**). The notation $(i_1,j_1) <_{[f,b]} (i_2,j_2)$ is supposed to indicate that (i_1,j_1) is eliminated either forward or backward depending on the condition

$$\texttt{ffi}\{(i_1,j_1)\} \leq \texttt{bfi}\{(i_1,j_1)\}.$$

Operations count based heuristics

The minimal number of multiplications required for the elimination of an edge (i,j) is the minimum out of the number of predecessors of its source and the number of successors of its target. Thus, we can formulate a heuristic equivalent to the **V_LM** heuristic. The **Lowest-Markovitz-Degree-First** edge elimination heuristic will be denoted by **E_LM**:

$$w_k \in M : \quad (i_1,j_1) <_{[f,b]} (i_2,j_2)$$

$$\Leftrightarrow$$

$$\min\{|P_{i_1}|_k, |S_{j_1}|_k\} \leq \min\{|P_{i_2}|_k, |S_{j_2}|_k\}.$$

Again, (i_1,j_1) is eliminated either forward or backward depending on the condition $|P_{i_1}|_k \leq |S_{j_1}|_k$.

Dependency degree based heuristics

As in the case of vertex elimination we can consider the minimal number of multiplications required for the elimination of an edge (i,j) relative to the actual input-[output-]dependency degree of v_i [v_j]. This leads to the **Lowest-Relative-Markovitz-Degree-First** edge elimination heuristic (**E_LR**):

$$w_k \in M : \quad (i_1,j_1) <_{[f,b]} (i_2,j_2)$$

$$\Leftrightarrow$$

$$\max\{\texttt{id}_{i_1} - |P_{i_1}|_k, \texttt{od}_{j_1} - |S_{j_1}|_k\} \geq \max\{\texttt{id}_{i_2} - |P_{i_2}|_k, \texttt{od}_{j_2} - |S_{j_2}|_k\}.$$

Notice that the dependency degree of a vertex is not invariant with respect to the general edge elimination method and has therefore to be recalculated at every stage in the metagraph. However, we could do the calculation just once and consider the vertex degrees relative to these initial dependency degrees.

As for the vertex elimination heuristics we have presented one representative of each class. Building on the above observations we can construct numerous variations by introducing different tie-break criterions or by altering certain parameters. In fact, none of the presented heuristics for edge elimination ever lead to a lower operations count for the accumulation of the Jacobian then the best known vertex elimination heuristic when applied to our selection of test problems.

3.3 Case Study

3.3.1 Absorption function

Consider a function of the form $y = f(x)$ with $f : \mathbb{R}^n \to \mathbb{R}$ and

$$y = \prod_{i=0}^{n-1} \left(\varphi_i(x_i) \cdot \prod_{j=1}^{e_i} x_i \right) .$$

A special case is given by

$$y = f_a(x) = \prod_{i=0}^{n-1} \left(\cos\left(x_i + i(2) \cdot \frac{\pi}{2} \right) \cdot \prod_{j=1}^{2(i+1)} x_i \right)$$

where $i(2) \equiv i \mod 2$ and which we will refer to as the *absorption function*. This name is motivated by the shape of the c-graph which is shown in Figure 3.2 and the behavior of f_a with respect to different vertex elimination strategies. It represents an example where as a result of the absorption of potentially new edges the backward elimination mode is not necessarily optimal in terms of the number of multiplications required for the computation of a gradient. Looking at the c-graph of f_a we observe the following: The backward vertex elimination mode starts with the elimination of the $n-2$ multiplications performed in the outer loop at a Markovitz degree of two each. It goes on with the $2(i+1)$ multiplications from the inner loop, again, each of them at a cost of two followed by the n univariate scalar functions φ_i. Hence,

$$\mathbf{OPS}_{\mathbf{V\_B}}\{CG_{f_a}\} \quad = \quad 2(n-2) + 2\sum_{i=0}^{n-1} e_i + n.$$

Using the forward vertex elimination strategy we would start with the φ_i followed by the elimination of all multiplications of the inner loop at a Markovitz degree of one due to the exploitation of absorption. Finally, the outer loop multiplications are eliminated successively with their costs incremented by one after each step:

$$\mathbf{OPS}_{\mathbf{V\_F}}\{CG_{f_a}\} \quad = \quad \binom{n}{2} - 1 + \sum_{i=0}^{n-1}(e_i + 1).$$

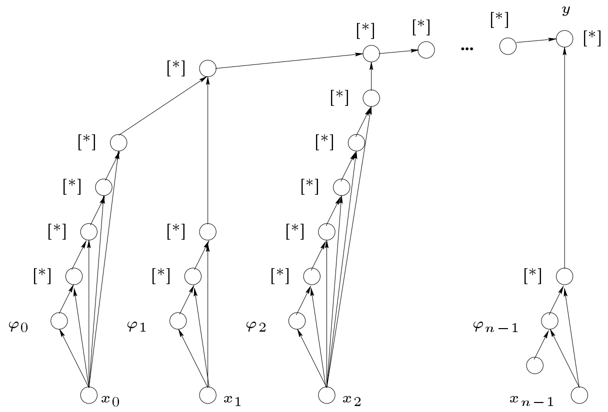

Figure 3.2: Absorption Function

Obviously, we would prefer some combination of the forward and backward elimination modes in order to be able to determine the minimal cost for accumulating the gradient of f_a. This is what our **V_LR** heuristic does. It makes full use of absorption as the forward mode does while avoiding the incrementation of the vertex degrees by running through the outer loop backward:

$$\mathbf{OPS}_{\mathbf{V\_LR}}\{CG_{f_a}\} \quad = \quad 2(n-2)+\sum_{i=0}^{n-1}(e_i+1).$$

For our special case we have with $e_i = 2(i+1)$:

$$\begin{aligned}
\mathbf{OPS}_{\mathbf{V_B}}\{CG_{f_a}\} &= 2n^2+5n-4,\\
\mathbf{OPS}_{\mathbf{V_F}}\{CG_{f_a}\} &= \frac{3}{2}n^2+\frac{3}{2}n-1,\\
\mathbf{OPS}_{\mathbf{V_LR}}\{CG_{f_a}\} &= n^2+4n-4.
\end{aligned}$$

We observe that with increasing values of n **V_B** becomes better compared to **V_F** while never reaching the same low cost. Moreover, neither forward nor backward vertex elimination is optimal with respect to the number of multiplications required for the accumulation of the gradient of f_a. As n increases their costs converge to a fixed multiple of the value of the optimum.

As for the *belt function* we have applied the verified global optimization algorithm which is described in [Nau94] to f_a. Both functions seem to be suitable as test problems for algorithms for unconstrained optimization.

3.3.2 Discretization of evolutionary equations

Suppose we have a local evolution process on a circle so that after discretization

$$z_{t+1} = C_t'(z_t) \quad \text{for} \quad t = 0, 1, \ldots, l-1,$$

where $z_t \in I\!R^\eta$ describes the state at time t at η points on the circle. Assuming differentiability and locality we find that the local Jacobians $C'(z_t) \equiv C_t'$ exist and

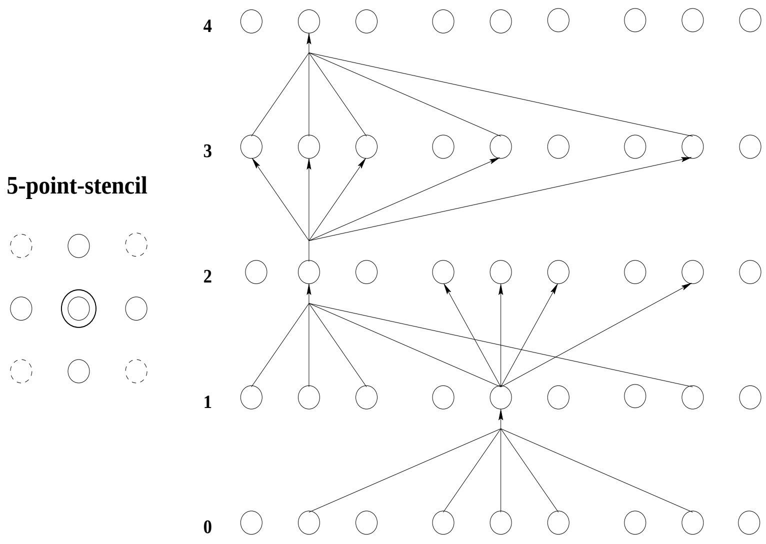

Figure 3.3: Evolutionary Equations

are bounded. Without loss of generality we assume that the C'_t are tridiagonal with a nonzero $(1, \eta)$ and $(\eta, 1)$ element, i.e. cyclic. Let us consider the task of computing

$$\frac{\partial z_l}{\partial z_0} = C'_{l-1}C'_{l-2}\cdots C'_1C'_0$$

with a minimal number of operations. The forward and backward methods would calculate according to the parenthesizations

$$C'_{l-1}(C'_{l-2}\cdots(C'_2(C'_1C'_0))\cdots)$$

and

$$(\cdots((C'_{l-1}C'_{l-2})C'_{l-3})\cdots C'_1)C'_0,$$

respectively. Now let us examine the question whether $\partial z_l/\partial z_0$ can be obtained cheaper (or more expensive) by other methods, i.e. parenthesization schemes. A good contender would seem to be the Markowitz compatible scheme

$$((C'_{l-1}C'_{l-2})(C'_{l-3}C'_{l-4}))\cdots((C'_3C'_2)(C'_1C'_0)). \tag{3.2}$$

We can show that in this case Markowitz is in fact optimal and slightly more efficient than forward or backward in terms of the operations count.

Now, suppose we consider a regular cube of $(1+\eta)^d$ points in $\mathbb{R}^d$. Assuming periodicity of the nodal states we only have to consider η^d points. If the next state in each point depends only on the nearest neighbors in the l_∞−norm (as in the one dimensional case considered first) the Jacobians C'_t have $\eta^d \cdot 3^d = (3\eta)^d$ nonzero entries. The product $C'_{t+1}C'_t$ has $(5\eta)^d$ nonzero entries which represents a cube of half width 2 rather than 1.

It is possible to show that for the multidimensional situations Markowitz is definitely not the right way to go. This is supported by our test results. We have applied several heuristics to c-graphs resulting from the application of a five-point stencil to a two-dimensional problem. Figure 3.3 shows the dependency scheme of the five-point stencil in two dimensions with the vertices drawn with dashed lines not depending on the locally dependent vertex in the center. On the right hand side we have displayed a c-graph for a two-dimensional problem with $\eta = 3$ and $l = 5$.

While both **V_F** and **V_B** eliminate the vertices in CG level by level **V_LM** performs several sweeps through the graph always eliminating every second level. This behavior leads to a strong increase of the Markovitz degrees of the remaining vertices with every traversal. Referring to the example graph in Figure 3.3 we observe that there are 3 intermediate levels to eliminate. The optimal strategies (e.g. **V_F** and **V_B**) would eliminate one level at a cost of 25 multiplications per vertex followed by the remaining two levels at $9 \cdot 45$ scalar multiplications resulting in a total cost of 1035. **V_LM** with a simple forward approach as primary tie-breaker would start with the first level $(9 \cdot 25)$ followed by the third at the same cost and, finally, it would have to eliminate the vertices at the second level at a cost of $9 \cdot 81$. This process would take 1179 scalar multiplications. The following table lists the values for a selection of heuristics applied to the simple graph in Figure 3.3. **V_HM** stands for a **Highest-Markovitz-Degree-First** approach which represents an upper bound for all our heuristics in most cases. It is supposed to give an impression of "how bad" it may get.

V_B	V_LM	V_HM	V_RPL	V_LR	E_LF	E_LM
1035	1179	1869	1035	1035	1421	1190

The above example is supposed to illustrate the problem and give a first idea of how different heuristics would behave. Both **V_RPL** and **V_LR** are still optimal in this case. Below we have listed the values for a slightly larger problem with $\eta = 4$ and $l = 9$ resulting in a c-graph which contains 128 intermediate vertices and 720 edges. While **V_RPL** is still optimal this is not true for **V_LR** anymore. In general, we observe that **V_RPL** is optimal on discretizations of evolutionary equations whereas all Markovitz related heuristics are not.

V_F	V_LM	V_LR	V_RPL
8880	11152	10160	8880

Chapter 4

Dynamic Programming

4.1 Roots of our Approach

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.

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

$$A = A_1 \cdot A_2 \cdot \ldots \cdot A_p \tag{4.1}$$

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

$$\Pi(k) \quad = \quad \begin{cases} 1 & \text{if } k = 1 \\ \sum_{l=1}^{k-1} \Pi(l)\Pi(k-l) & \text{if } k \geq 2 \end{cases}, \tag{4.2}$$

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 (4.1) 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 4.1 and 4.2. Consider the computation of an optimal parenthesization of a product of four matrices $A_1, \ldots, A_4$. With Equation (4.2) 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 4.1 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 4.1 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

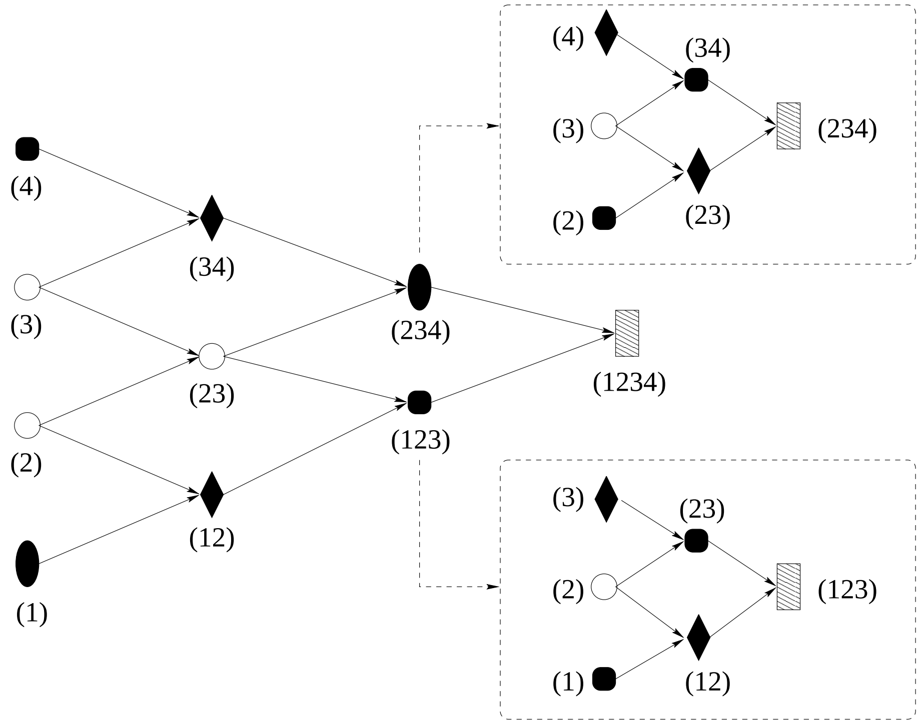

Figure 4.1: Table Look Up Graph

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,

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

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

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

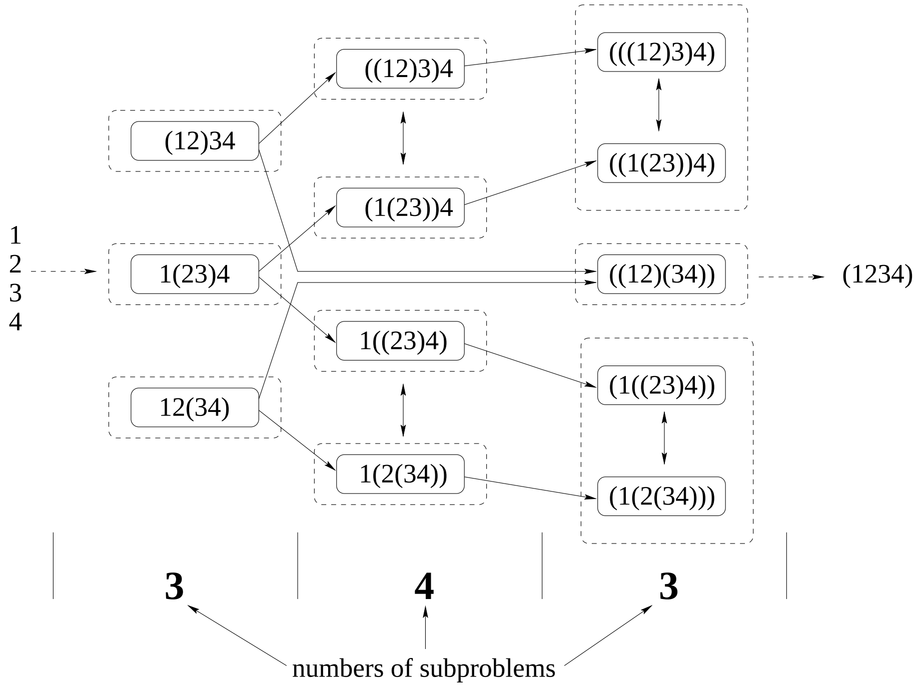

Figure 4.2: Metagraph

vertices and

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

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 4.2 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 polynomial in the number k of factors in the chained matrix product ($O(k^3)$).

4.2 Jacobians as Chained Matrix Products

With the terminology introduced in Chapter 1 we have 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

$$J = Q_m(C_{p+m} + I - \mathbf{e}_{p+m}\mathbf{e}_{p+m}^T) \ldots (C_1 + I - \mathbf{e}_1\mathbf{e}_1^T)P_n^T. \qquad (4.3)$$

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:

$$J = Q_m(\bar{C}_p + I - \mathbf{e}_p\mathbf{e}_p^T)\ldots(\bar{C}_{1-n} + I - \mathbf{e}_{1-n}\mathbf{e}_{1-n}^T)P_n^T. \tag{4.4}$$

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

$$B_j = C_j + I - \mathbf{e}_j\mathbf{e}_j^T \quad \text{and} \quad \bar{B}_j = \bar{C}_j + I - \mathbf{e}_j\mathbf{e}_j^T \tag{4.5}$$

we get

$$J = Q_m(B_{p+m}\ldots B_1)P_n^T \qquad \textbf{scalar tangent chain,} \tag{4.6}$$

$$J = Q_m(\bar{B}_p\ldots\bar{B}_{1-n})P_n^T \qquad \textbf{scalar adjoint chain}. \tag{4.7}$$

Regarding the C_j's as *additive* local extended Jacobians as in Section 1.2 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.

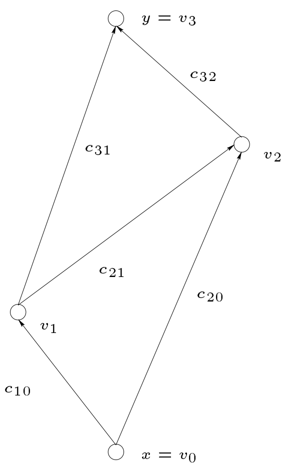

Figure 4.3: B_2B_1 (1)

We could 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 solving the shortest path problem on the corresponding subgraph of the metagraph. We have that 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 we have

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

$B_{[j]}$ $[\bar{B}_{[j]}]$ 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

$$i \sim j \qquad \Leftrightarrow \qquad i \not\prec^* j \quad \text{and} \quad j \not\prec^* i. \tag{4.8}$$

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 1.3. 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 4.2 and 4.1 have to be reversed when regarding edge elimination in c-graphs. Simply set

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

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

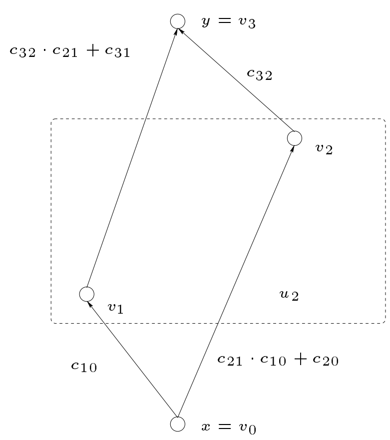

Figure 4.4: $B_2 B_1$ (2)

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 4.2 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 multiplication 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 is exponential in 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 4.3 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 4.4). 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 mutually unreachable elemental vertices from CG. We will introduce quotient graphs in the next section.

4.3 Computational Quotient Graphs

Definition 4.3.1 *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 (4.8) 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 (4.5) 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:

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

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] Markovitz degree of a superedge $([i],[j])$ analogous to the general vector mode which was introduced in Section 1.3, i.e.

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

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 1.3 where the general vector elimination mode was introduced. In $CG/\sim$ every supernode u_i can be associated with a vector operation

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

with d_i^{pred} and d_i defined as at the beginning of Section 1.2.1. As in the general vector mode we can associate local Jacobians

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

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.

At the beginning of Chapter 2 we have listed a set of restrictions on the metagraph which reduce the size of the search space of the shortest path problem. Apart from the chained matrix product approach we have also mentioned both edge and vertex elimination in quotient graphs. Obviously, both of them represent more general approaches to our original problem at reduced but still exponential subgraphs of the metagraph.

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

Definition 4.3.3 *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 4.3.1 *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 4.3.4 *Let $\mathcal{I} : V(CG) \rightarrow \mathbb{N}$ 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.*

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 4.3.2 *For a given consistent numbering $\mathcal{I} : V(CG) \rightarrow \mathbb{N}$ 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 subminimal 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 4.5. Analogous, a *Breadth First Reverse* (BFR) strategy also delivers a well-defined numbering of all vertices in CG which corresponds to Definition 1.1.2. In general, the numberings, resulting from the application of BFF and BFR are not the same.

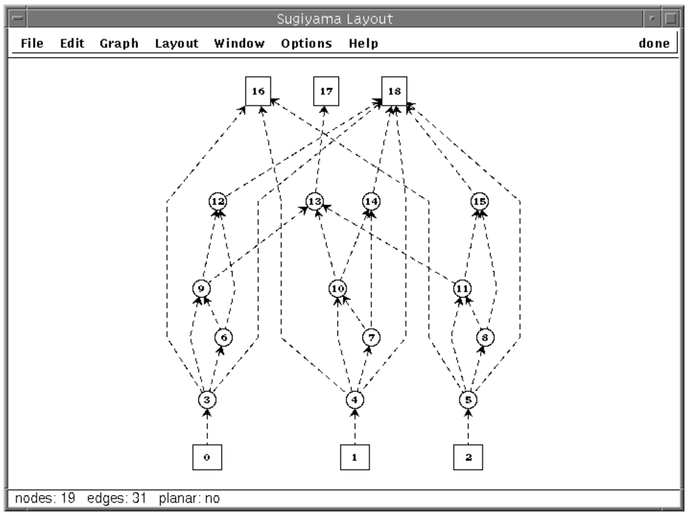

Figure 4.5: Chebyshev Quadrature, $n = m = 3$, BFF

4.4 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 Section 6.1 or have a look at the example session in Appendix A. 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 $CG/\sim$.

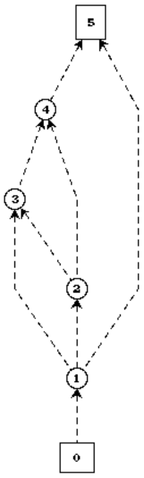

Figure 4.6: $CG/\sim$

Apart from the c-graph which is shown in Figure 4.5 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 4.6.

Figure 4.7 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 in T interactively and visualize their characteristics as shown in Figure 4.7 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.

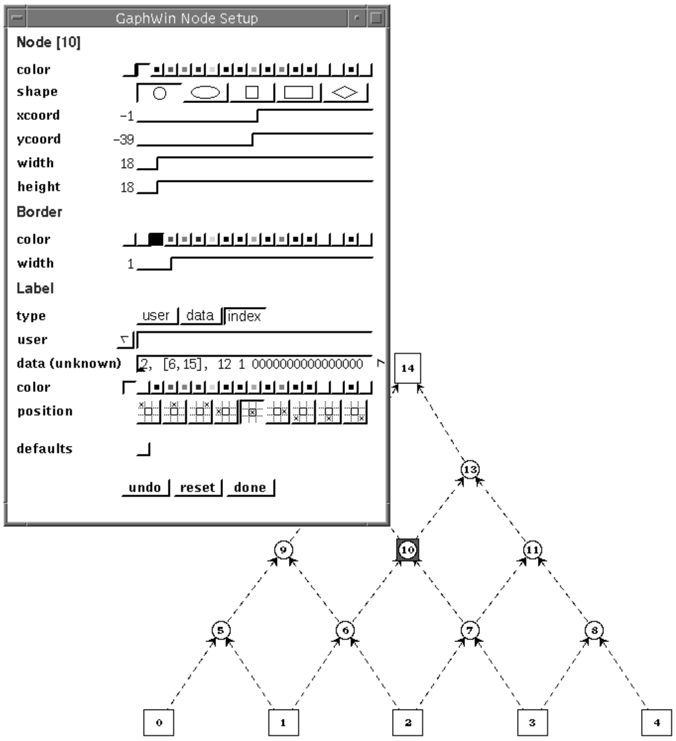

Figure 4.7: Table Look Up Graph

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.

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.

In Chapter 6 we will present a description of the software **OESCOMP** and numerous test results including comparisons between different heuristics, the dynamic programming approach, and a simulated annealing algorithm which we will describe in the next chapter.

Chapter 5

Simulated Annealing

5.1 SAVE

In this chapter we will present the idea behind the abbreviation **SAVE** which stands for **S**imulated **A**nnealing applied to the **V**ertex **E**limination problem in c-graphs. 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 general, combinatorial optimization problems are characterized by an objective function to be minimized (the overall Markovitz degree in our case) and a discrete, very large configuration space (the $p!$ different orders in which we can eliminate the p intermediate vertices in the c-graph). Furthermore, the concept of *direction* telling us about *going downhill* or not does not exist in our context.

The roots of the simulated annealing algorithm lie in metallurgy. During the annealing process of metal its atoms form crystals. An optimal crystal is one with a configuration of lowest energy E. If we run the cooling process too fast then the crystals will not reach the optimal configuration. On the other hand, we do not want the annealing process to take too long. The goal is to find a useful compromise between the robustness of the algorithm and its speed. This is done by varying certain parameters within the algorithm.

Let us assume that we are looking for the configuration which minimizes a certain cost function. The algorithm can then be formulated as follows: Starting off at an initial configuration, a sequence of iterations is generated. Each iteration consists of the random selection of a configuration from the neighborhood of the current configuration and the calculation of the corresponding change in the cost function. The neighborhood is defined by the choice of a generation mechanism, i.e. a "prescription" to generate a transition from one configuration into another by a small perturbation. If the change in the cost function is negative the transition is unconditionally accepted. If the cost function increases the transition is accepted with a probability based upon the Boltzmann distribution

$$P(E) \quad \sim \quad e^{\frac{E}{k \cdot T}}$$

where k is a constant and the temperature T is a control parameter. This temperature is gradually lowered throughout the algorithm from a sufficiently high starting value (i.e. a temperature where almost every proposed transition, both positive and negative, is accepted) to a "freezing" temperature where no further changes occur. In practice, the temperature is decreased in stages and at each stage the temperature is kept constant until thermal quasi-equilibrium is reached. The whole of parameters determining the temperature decrement (initial temperature, stop criterion, temperature decrement between successive stages, number of transitions for each temperature value) is called the cooling schedule.

Consequently the four key ingredients for the implementation of simulated annealing are:

1. the definition of configurations of the system;

2. a generation mechanism, i.e. the definition of a neighborhood on the configuration space; a generator of random changes in the configuration;

3. the choice of a cost function (analog of energy) the minimization of which is the goal of the procedure;

4. a control parameter T and a cooling schedule telling us after how many random changes in configuration each downward step in T is taken. The assignment of this schedule requires deeper insight into the problem and trial-and-error experience.

5.1.1 The traveling salesman problem

In [PTVF92] we have found a concrete illustration of how to approach combinatorial optimization problems by simulated annealing. The *traveling salesman problem* (TSP) belongs to the class of NP-complete problems whose computation time for an exact solution grows exponentially with its size. For an exhaustive discussion of NP-completeness refer to [GaJo79].

The TSP may be regarded as the "mother" of all transport optimization problems. A salesperson visits n cities with given coordinates (x_i, y_i), returning finally to his city of origin. Each city is to be visited only once while making the route as short as possible. This leads to the objective function

$$E = \sum_{i=1}^{n} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \qquad ((x_{n+1}, y_{n+1}) \equiv (x_1, y_1)) \tag{5.1}$$

which is taken just as the total length of the journey. The above is equivalent to finding a *Hamilton cycle* of minimal length in the complete graph K_n with edges representing the distances between the cities. The vertex elimination problem in a c-graph is very similar to the *TSP*. Thus, many algorithms that work well for the latter can often easily be adapted to serve our purposes.

5.1.2 Application to the vertex elimination problem

As a problem in simulated annealing the vertex elimination problem is handled as follows:

1. Configuration. The vertices are numbered $i = 1, \ldots, p$ and each of them has a Markovitz degree $|P_i|_k \cdot |S_i|_k$ depending on the stage w_k in the metagraph. A configuration is a permutation of the indices $1, \ldots, p$ interpreted as the order in which the intermediate vertices are eliminated. The key difference between our problem and the traveling salesman problem is that the Markovitz degree is not a static value. It rather depends on the vertices that have been eliminated so far. However, in the TSP every city has its static coordinates making this problem even "easier" to solve. (This is not to be taken literally as the TSP is proven to be NP-hard ...)

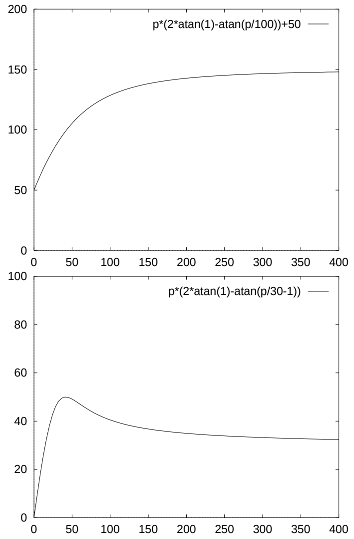

Figure 5.1: Loops

2. Rearrangements. We use a slightly altered version of the moves suggested by Lin ([Lin65]). There are two types of moves:

1. If $(i_1, \ldots, i_p)$ is the current elimination sequence then we remove a dense subsequence $(i_j, \ldots, i_{j+k})$ and replace it with itself in the opposite order making

$$(\ldots, i_{j-1}, i_{j+k}, \ldots, i_j, i_{j+k+1}, \ldots)$$

the next elimination sequence to be regarded or

2. a dense subsequence $(i_j, \ldots, i_{j+k})$ is removed and then replaced in between two indices i_l, i_r on another, randomly chosen, part of the elimination sequence, i.e. $(i_1, \ldots, i_p)$ becomes

$$(\ldots, i_{j-1}, i_{j+k+1}, \ldots, i_l, i_j, \ldots, i_{j+k}, i_r, \ldots)$$

provided that $l > j + k$. Otherwise, we get

$$(\ldots, i_l, i_j, \ldots, i_{j+k}, i_r, \ldots, i_{j-1}, i_{j+k+1}, \ldots)$$

for $r < j$. We do not permit other rearrangements apart from these two.

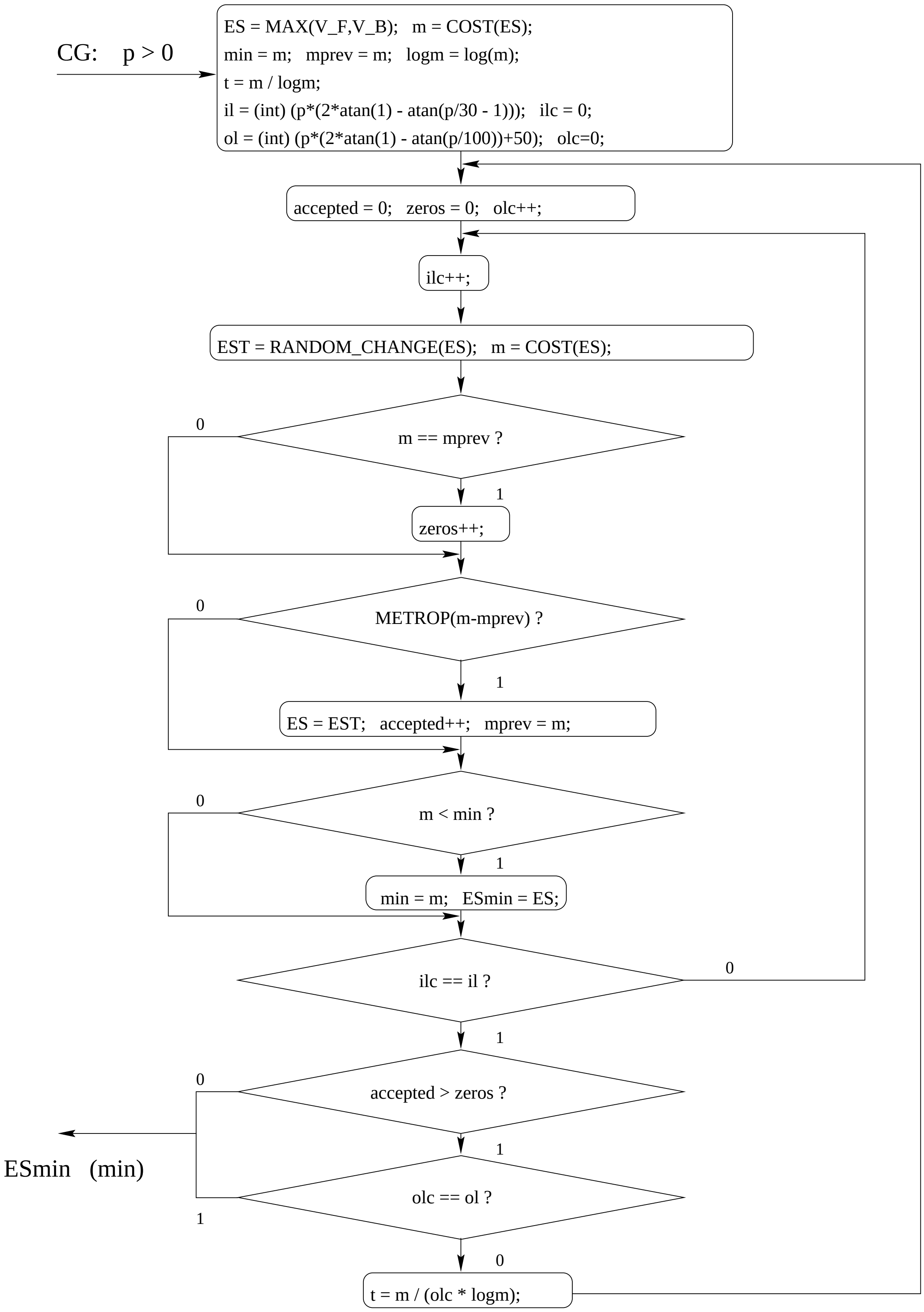

Figure 5.2: Simulated Annealing Algorithm

3. Objective function. We intent to minimize the cost of accumulating the complete Jacobian matrix of a vector function by solving the shortest path problem in the metagraph $M_{\mathbf{V}}$ which is induced by the restriction to pure vertex elimination. Since the Markovitz degree of a vertex is dynamically changing throughout the elimination process its calculation is rather expensive ($O(p)$) having in mind that we will have to compute it numerous times during the simulated annealing algorithm. Notice that the computation of the Euclidean distance between two cities in the TSP which is part of Equation (5.1), is comparably cheap.

4. Annealing schedule. Figure 5.2 shows how the implemented simulated annealing algorithm works. It starts with an initial elimination sequence (`ES`) which we have chosen to be the maximum out of forward and backward vertex elimination modes in terms of the number of multiplications required for the accumulation of the complete Jacobian. At the beginning of the optimization procedure we do not want to be the cost `m` of `ES` close to the minimum. In this case it would become very likely that the algorithm stops after just one iteration without delivering any improvement.

Our simulated annealing algorithm consists of two main loops – an outer loop (counter: `olc`) and an inner loop (counter: `ilc`). For both of them we define upper bounds for the number of iterations they will perform (`ol` and `il`). `il*ol` is the maximal number of iterations (the maximal number of elimination sequences that are generated and run) before the algorithm stops, even if it has not converged. It may happen that one and the same elimination sequence is checked twice or even more often, although it is very unlikely. We make sure that we get a result within a reasonable time span by setting `ol` and `il`. Its quality strongly depends on the parameters of the simulated annealing algorithm.

Figure 5.1 shows the development of `il` and `ol` depending on the number `p` of intermediate variables in the c-graph CG. Notice that for our algorithm we always assume that there is at least one vertex to be eliminated from CG. The maximal number of cooling steps lies always between 50 and 150. It increases rapidly for small values of `p`, whereas it converges to 150 for larger graphs. Analogous, we observe a steep ascent in the first part (`0<p<50`) of the curve for `il`. Again, it settles for a value around 40 for increasing numbers of intermediate variables. Choosing both values as we did assures that the algorithm will always terminate after a reasonable and in most cases sufficient run-time.

The "temperature" `t` is lowered with every iteration of the outer loop and it is kept constant while running through the inner loop. Within the latter we randomly change the elimination sequence using the two *rearrangements* suggested above. Whether this new, so far temporary, order (`EST`) is accepted or not depends on the Metropolis criterion. It returns a boolean variable which issues a verdict on whether to accept a reconfiguration leading to a change `d=m-mprev` in the objective function `COST`. If `d<=0 EST` will always be accepted. If `d>0` the answer of `METROP(d)` will only be positive with probability `exp(-d/t)`. As we have already pointed out, `t` is not changed within the inner loop. However, with every iteration (`olc`) of the outer loop we reinitialize `t` as `t=m/(olc*logm)`. Notice that the new value of `t` depends on the current Markovitz degree, i.e. it depends on the cost of the last accepted elimination sequence. With this approach we get

$$t_j = \frac{m_{j-1}}{(j-1) \cdot \log(m_0)} \quad \Rightarrow \quad t_{j+1} = \frac{m_j}{j \cdot \log(m_0)}.$$

From the above we can derive the **cooling rate** Δ_t:

$$\Delta_t = \frac{t_{j+1}}{t_j} = \frac{(j-1) \cdot m_j}{j \cdot m_{j-1}}.$$

Why have we chosen to let `t` develop this way? Suppose we have taken a step upwards (accepted an elimination sequence which results in an increase of the objective function), i.e. $m_j > m_{j-1}$. Then we do not want to continue the cooling process at the same speed as before since this could lead to being unable to leave the current "valley". So, the temperature is permanently lowered as a result of the increasing outer loop counter `olc`. However, the extend up to which the system is cooled depends both on the current temperature and on the change of the Markovitz degree during the last iteration. This approach turned out to work well in most cases (see Section 6.2).

Throughout the entire annealing process we always keep track of the minimal Markovitz degree (`min`) resulting from any of the elimination orders that were checked so far. Thus, we do not entirely depend on the convergence of the algorithm. Even a very good elimination sequence which was found "by luck" in the high temperature phase of the annealing process can be the result of running the algorithm. After each outer loop iteration we check the *exit criterions*. There are three of them:

1. The maximal number of iterations to be performed is reached (`olc==ol`).

2. During the last outer loop iteration none of the rearrangements has been accepted.

3. We have chosen to accept all rearrangements which do not lead to any change in the cost function. However, if the only accepted rearrangements are those which are invariant with respect to the cost (`accepted == zeros`) we will also stop the algorithm.

Finally, the simulated annealing algorithm delivers the elimination sequence with the minimal cost, i.e. a vertex elimination order which approximates the minimal number of multiplications required for the accumulation of the complete Jacobian.

Summarizing the above we have the following list of parameters which can be experimented with when looking for an optimal annealing schedule:

- initial elimination sequence;
- initial temperature;
- cooling rate;
- types of rearrangements;
- number of iterations with constant temperature;
- number of cooling steps;
- acceptance philosophy.

There is certainly plenty of room for experiments. We have implemented four additional versions of our simulated annealing algorithm. In **SA_R** we use the reversal of dense subsequences as the only rearrangement action. **SA_T** is similar except for only transportation of dense subsequences being allowed. Furthermore, we have experimented with two different annealing schedules. **SA_CS** is similar to the described method with

$$t_j = \frac{m_{j-1}}{2j} \quad \Rightarrow \quad \Delta_t = \frac{t_{j+1}}{t_j} = \frac{j \cdot m_j}{(j+1) \cdot m_{j-1}},$$

i.e. we have slowed the cooling process down. In **SA_CF** we increase the temperature in the second step, which can be regarded as the generation of a new random initial elimination sequence, as we accept almost every rearrangement. Then we cool the system down with

$$t_j = \frac{m_{j-1}}{j^2} \quad \Rightarrow \quad \Delta_t = \frac{t_{j+1}}{t_j} = \frac{j^2 \cdot m_j}{(j+1)^2 \cdot m_{j-1}}$$

which is much faster then in the main method. In the algorithm described by Figure 5.2 we have used the following symbolism:

`ES, EST, ESmin` $\in \mathbb{N}^p$:	elimination sequences;
`m, mprev, min` $\in \mathbb{N} \cup \{0\}$:	cost values;
`il, ilc, ol, olc` $\in \mathbb{N} \cup \{0\}$:	for loops;
`t` $\in \mathbb{R}$:	temperature;
`logm` $\in \mathbb{R}$:	logarithm of initial cost;
`accepted, zeros` $\in \mathbb{N} \cup \{0\}$:	counters.

Now, it should be straight forward to follow the logic behind our simulated annealing algorithm.

5.2 Case Study

The success of simulated annealing depends on extensive tests and variation of parameters. Of course, we would like the algorithm to deliver nearly optimal results in virtually all cases without always having to adapt the parameters to the given problem. Furthermore, it should not run too long even for large-scale problems. We have applied **SAVE** to many test problems out of which we will discuss a small subset in detail. As supported by the results achieved (see Section 6.2) our version of the simulated annealing algorithm performs well on a large number of problems. However, there are certain exceptions indicating that it is impossible to come up with a strategy (annealing schedule) which is universally optimal.

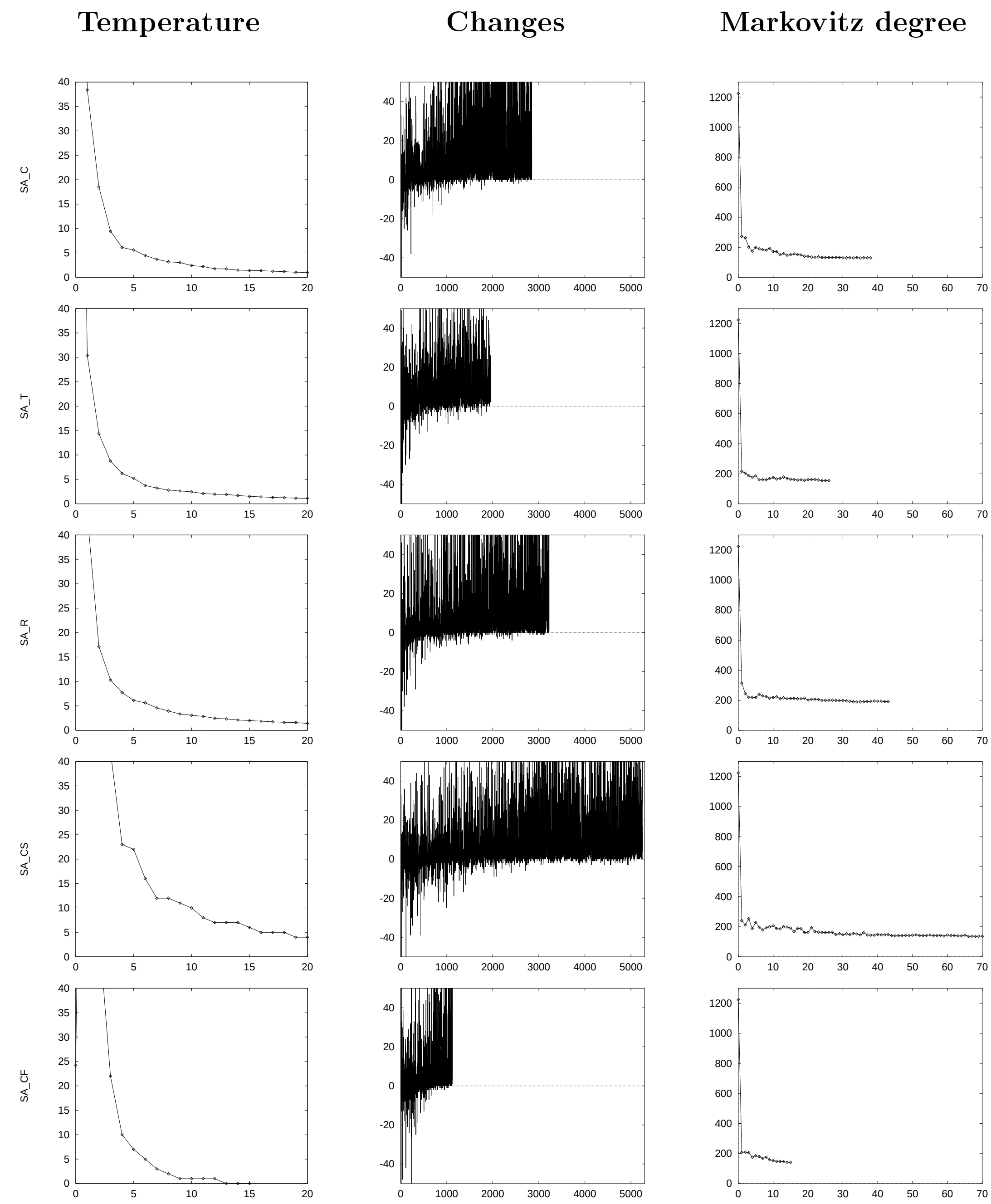

Figure 5.3: **SAVE** on SPF

Speelpenning function

The first test problem is due to Speelpenning [Spe80] and represents a simple chained product of the n independent variables:

$$y = f(\mathbf{x}) = \prod_{i=0}^{n-1} x_i. \qquad \text{(SPF)}$$

For $n = 50$ the c-graph contains 48 intermediate vertices. It can be checked that **V_B** is optimal on this problem using 96 multiplications for the computation of the gradient of f. On the other hand, running **V_F** results in an overall Markovitz degree of 1224 as a consequence of the successive increase of the in-degrees of the intermediate vertices.

Using Figure 5.3 we will compare all variations of our simulated annealing algorithm which form the five rows of the table. The three columns show the development of the temperature over the first 20 steps, the changes in the objective function after each iteration, and the course of the overall Markovitz degree which we intent to minimize, respectively. We have plotted the values for the temperature and the Markovitz degree with respect to the outer loop iterations, whereas the changes in the cost are shown for every single elimination sequence that has been checked, i.e. for each iteration of the inner loop. Apart from differences in the run-time the five methods converged to different final values of the overall Markovitz degree:

	SA_C	**SA_T**	**SA_R**	**SA_CS**	**SA_CF**
`min`	129	154	189	136	142
$t_{\texttt{user}}$	93 sec	63 sec	105 sec	171 sec	37 sec

Furthermore, we observe the following:

1. Neither a higher nor a lower cooling rate lead to improvements in the value of the objective function. However, the run-time of **SA_CF** undercuts the one of **SA_C** by a factor of nearly 3. Still it delivers an acceptable result.

2. Starting with the **V_F**-based elimination sequence, the reversal of substrings could be expected to lead to a good solution in a short time. Obviously, this is not the case. Why? The reason lies in the structure of the Speelpenning function. Whenever we eliminate the vertices of a part of the c-graph backward this can be regarded as "good" for the value of our objective function. However, running forward is certainly bad. Now, if we allow the reversal of sub-strings of a given elimination sequences to be the only rearrangement in the annealing process this can lead to the repeated negation of savings made in the current step by the rearrangements to come.

3. Depending on the accepted rearrangements and the resulting Markovitz degree the temperature is lowered at varying speeds. Especially, this becomes clear when looking at the graphs for **SA_CS** in the fourth row of Figure 5.3.

4. The largest decreases in the objective function value are achieved in the high temperature phase. This is certainly not very surprising.

5. It might often be advantageous to choose a high cooling speed in order to get useful estimates for the savings that can be expected.

It is difficult to state something of general validity when speaking about the method of simulated annealing applied to computationally hard combinatorial optimization problems. However, the approach that we have chosen as our main method (**SA_C**) turned out to deliver the best results in virtually all cases.

Steady state combustion problem

The steady state combustion problem (SSC) is the variational formulation of the underlying boundary value problem [ACM91]. For the examination of the behavior of simulated annealing we have chosen the case with $n = 4$ independent variables. This leads to a relatively small c-graph containing 103 intermediate vertices which is suitable for having a closer look at our algorithm.

SA_C converged to an elimination sequence that took 123 multiplications to compute the gradient. The **V_B**-based sequence (142 multiplications) served as the starting point. The best known value of the objective function achieved by **V_LR** as well as the dynamic programming approach (see Chapter 4) and the method **SA_CS** is 122. Figure 5.4 shows the courses taken by the temperature t and by the Markovitz degree m_j.

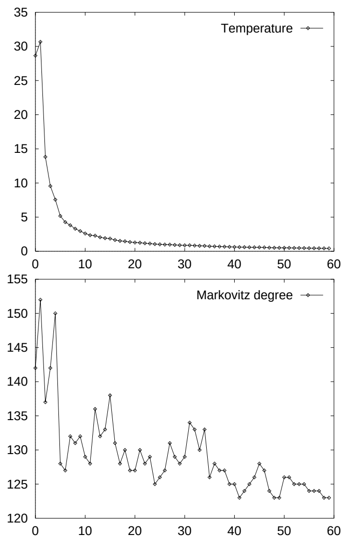

Figure 5.4: SSC

With `il=47` and `ol=130` the algorithm had to terminate after checking 6110 vertex elimination sequences. In fact, it performed only 59 outer loop iterations corresponding to the generation of 2773 (not necessarily different) elimination orders after which one of the *exit criterions* described in Section 5.1 was met. Exploiting one of the advantages of our simulated annealing algorithm the Markovitz degree is increased repeatedly, thus, being able to escape from local minima.

Obviously, the savings which can possibly be achieved are not so remarkable that they could justify the effort. It makes sense to apply simulated annealing to the vertex elimination problem in c-graphs if we care about the result only and ignore the time that it took to compute the elimination sequences. In most cases heuristics will deliver sequences which are nearly as good (or better) at a much higher speed.

Chebyshev quadrature problem

Both the SSC and the Chebyshev quadrature problem (CQ) are taken from the `MINPACK` test problem collection [ACM91]. For the latter we have chosen the case $n = 15$ and $m = 16$ resulting in a c-graph of reasonable size. Remember that we always take either the **V_F**- or the **V_B**-based elimination sequence as an initialization of the simulated annealing algorithm, depending on which of them delivers the higher cost. Starting with a relatively "bad" initial elimination sequence turned out to be advantageous for the behavior of the algorithm in many cases. Now, this function represents a counterexample as the starting sequence is obviously "too bad" for a problem of the given size and structure. The operations count resulting from the application of the backward vertex elimination mode (8400) is about four times as large as the one of the forward mode (1980). The algorithm "cools the system down" to 4842 which is approximately half the number of multiplications compared to its starting value. However, it does not reached the value which was calculated for **V_F**.

Figure 5.5 shows the development of the Markovitz degree. After some ups and downs in the first section of the annealing process, caused by the initially high temperature, the cost decreases continuously, unfortunately, not reaching the minimum. As a way out we could think of slowing the cooling process down in order to allow more iterations to be performed. Representing this approach **SA_CS** results in a cost of 4343. In fact, we observe an improvement compared to **SA_C**, although it is not very encouraging when taking into account that it took about four times as long to achieve the improvement.

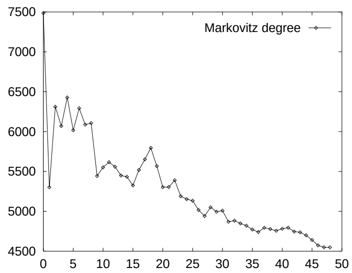

Figure 5.5: CQ

5.3 Conclusion

In Section 6.2 we have listed the results of running different algorithms for calculating optimized elimination sequences, including simulated annealing, on a selection of test problems. Simulated annealing delivered results near the best known in all cases. However, it could not undercut the minimum out of the values delivered by the other methods. Thus, we may conclude that, due to its high cost, simulated annealing will in general not be the number one choice for approaching the general edge elimination problem in c-graphs.

Chapter 6

Implementation and Tests

6.1 OESCOMP

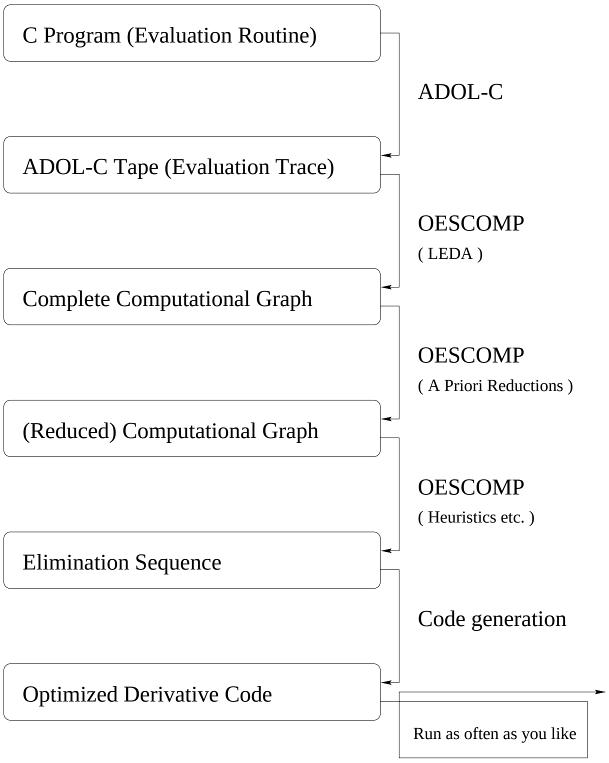

Figure 6.1: Scenario

The numerical results presented in the second part of this chapter were obtained using the program **OESCOMP** which represents the implementation of most of our theoretical results. The abbreviation stands for **COMP**utation of **O**ptimal **E**limination **S**equences. **OESCOMP** is an extensive software package consisting of several `C++` libraries the source codes of which add up to more than ten thousand lines. For better legibility we have used the `CWEB` system of structured documentation [LeKn90] which allows us to write software according to the principles of *literate programming* [Knu92]. In Appendix C we have included an excerpt of the `CWEB` output describing the implementation of the dynamic programming algorithms presented in Chapter 4. The `CWEB` software package consists of two parts which can be applied to the source file `filename.w` separately. `ctangle` is responsible for the conversion into a `C/C++` file which will then be subject to compilation. The TeX-file which contains the structured and documented source code is generated by `cweave`.

OESCOMP builds on `ADOL-C`, version 1.7, [GJU96] by using its internal representation of the evaluation trace (*tape*) for constructing the c-graph as a `LEDA` data structure [MeNä96]. The **L**ibrary of **E**fficient **D**ata Types and **A**lgorithms (`LEDA`) is developed at the Martin-Luther-University Halle and at the Max-Planck-Institute for Computer Science Saarbrücken. It provides a wide range of data structures such as lists, stacks, different number types, graphs, and numerous efficient algorithms to work with them. In our project we build on parameterized graphs implemented as templates (`GRAPH<nodetype,edgetype>`) and some related data types. For visualizing the directed acyclic graphs we have used an algorithm due to Sugiyama, Tagawa, and Toda [STT81] which is part of the `LEDA`-based `AGD`-library [AGD98].

The principal scenario of the implementation is shown in Figure 6.1. Starting with an evaluation routine, `ADOL-C` is used to break all statements down in order to get the required evaluation trace. **OESCOMP** performs all steps from extracting the c-graph to the computation of a nearly optimal edge or vertex elimination sequence. This information can be used to generate optimized adjoint code. However, this will be subject to further research (see Chapter 7) and there is no implementation of it, so far.

OESCOMP has a graphic user interface and it provides a wide range of functions for exploring differentiation problems. Its functionality includes features like

- chained matrix product optimization algorithms;
- heuristics for calculating elimination sequences;
- simulated annealing algorithms;
- a priori reductions of the c-graph;
- both vertex and edge elimination;
- visualization algorithms;
- information on the c-graph;
- several protocols.

We consider **OESCOMP** as a package for research and teaching which allows the user to gain deep insight into the structure of the problems. The program does not provide the possibility to accumulate Jacobians numerically, so far. It is merely supposed to enable the user to analyze the given problem and it can support the decision for a particular method. An example session discussing further details of the features provided by **OESCOMP** can be found in Appendix A.

We do not consider the current version of **OESCOMP** to be the final one. Building on what we have done so far, we plan to work constantly on improving the program and on extending its functionality. In Chapter 7 we will sketch the direction in which we intent to move as a part of our outlook, resulting from the research which led to this thesis.

6.2 Tests

6.2.1 Unsaturated flow problem

The first test problem which we will discuss more in detail is a two-dimensional unsaturated flow problem in a porous medium. An extensive description of it can be found in [CGRW92]. Its Jacobian J is an extremely sparse 1989×1989 square matrix yielding a very regular sparsity structure arising from the underlying discretization grid. We have chosen to optimize the computation of the sub-matrix $\hat{D} = J(0..935, 936..1286)$ of the Jacobian. The corresponding c-graph consists of 351 minimal, 936 maximal, and 21177 intermediate vertices. According to what we have introduced in Section 2.3 the approximate operations counts for the dense forward (**DF**) and for the dense reverse modes (**DR**) of AD are then equal to 350 thousand and 20 million multiplications, respectively. However, T. Robey [CGRW92] recognized that the 351 rows of $\hat{D}$ could be computed in only 6 passes. This would result in an remarkably decreased operations count of approximately 133 thousand in sparse forward mode, yielding an improvement by a factor of nearly 3.

By fully exploiting the sparsity of the problem in forward vertex elimination mode the cost could be decreased even further (73 thousand multiplications). The application of the cross-country vertex elimination method in form of the **V_LR** heuristic delivered the lowest known number of multiplications required for accumulating $\hat{D}$ (29 thousand), so far. A priori elimination of all hoisting vertices (see Section 2.5) led to a reduction of the number of intermediate vertices by 17433.

To summarize the above, we observe that the full exploitation of the sparsity leads to savings by a factor of 6. By solving the combinatorial problem of finding an optimal elimination sequence we could decrease this number by another factor of 2.5.

6.2.2 Further results

We have applied the cross-country elimination method to more than a hundred test problems out of which we will present a small but representative subset. Throughout this thesis we have already discussed some example problems with respect to the different methods, including heuristics, dynamic programming approaches, and simulated annealing algorithms. We will list a few results without presenting a detailed discussion for each of them below. 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 **V_LR** vertex elimination heuristic with the theoretical operations counts that can be achieved using state-of-the-art AD-technology (see Section 2.3). Here, $[\hat{n}, \hat{m}]$ denotes the minimum out of the maximal numbers of nonzero elements per row and per column in the Jacobian.

	n	p	m	$[\hat{n}, \hat{m}]$	**DF**	**DR**	**NR**	**V_LR**	**NR / V_LR**
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	504	**1.14**
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 **V_LR** vertex elimination mode 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.

The next table shows a comparison between the different approaches to the solution of the shortest path problem. We have also listed the values achieved by the better choice out of the forward and the backward vertex elimination modes, denoted as [**V_F**,**V_B**]. Notice that these results imply that we know which of the two methods is actually "the better choice". The differences between the results delivered by **V_F** and **V_B** can be very large in some cases.

	[V_F,V_B]	**V_LM**	**V_LR**	**V_LF**	**V_RPL**	**V_ELM**	**E_LF**	**BFRA**	**SA_C**
PDJ	1376	1278	1278	1409	1368	1376	1307	1278	1318
SSC	516	452	452	452	496	558	452	452	469
FDC	930	1338	930	1030	1002	1616	1106	930	1234
FCH	941	845	845	925	845	941	941	845	851
WAT	6473	4240	4240	4240	7488	7838	4240	4240	5183
DIE	2059	1659	1659	1864	1659	2001	2001	1659	1660
VDI	20400	10401	10301	10301	20201	15791	10301	10301	10337
PEN	1996	1798	1798	1857	1996	2011	1798	1798	1899
EXP	576	504	504	566	552	536	571	528	504
SPE	96	96	96	273	96	1224	273	96	129
GDF	1430	1430	1430	1528	1430	1625	1625	1430	1477

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 unidirectional 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.

Chapter 7

Conclusion and Outlook

7.1 Summary

The goal of this thesis 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 nonzero 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] Markovitz degree. The sum of the forward [backward] Markovitz degrees of all edges at the time of their elimination from the c-graph (overall Markovitz 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. According to Chapter 2 we conjecture that the vertex-edge discrepancy is less or equal to 2. Therefore, we have concentrated on vertex elimination for the development of heuristics for solving the shortest path problem in the metagraph.

We have presented numerous heuristics for eliminating both edges and vertices from c-graphs. Furthermore, 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. Simulated annealing methods turned out to be a very robust but expensive way to approach the vertex elimination problem. Most of our theoretical results were implemented and tested on numerous example problems.

7.2 Pros and Cons

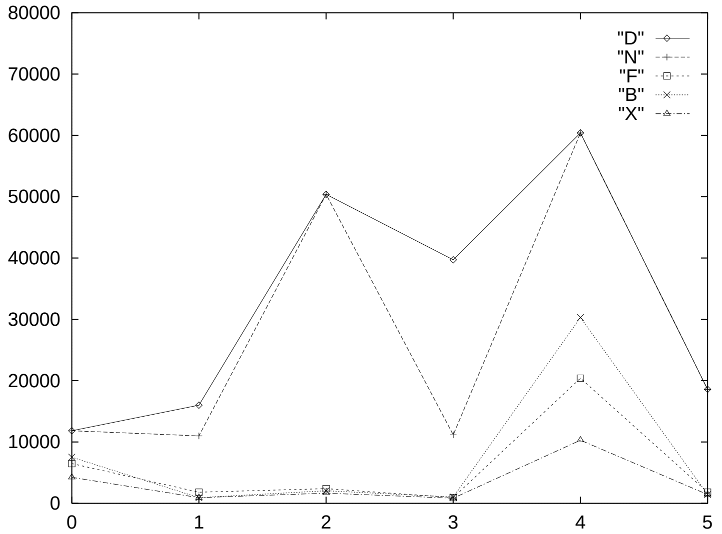

Figure 7.1: Different Approaches to AD (6 test problems)

One of the most significant features of the calculation of Jacobian matrices by edge elimination in c-graphs is its implicit exploitation of the sparsity of the given problem. This alone leads to savings by factors of 3 and more regarding the number of multiplications involved in the accumulation of the Jacobian for many problems. Obviously, this value cannot be reached for the calculation of pure gradients. Figure 7.1 is supposed to illustrate the superiority of vertex elimination. We have compared two approaches from state-of-the-art AD with the cross-country elimination method for 6 arbitrarily chosen test problems ($0 \hat{=}$ WAT, $1 \hat{=}$ FDC, $2 \hat{=}$ DIE, $3 \hat{=}$ FCH, $4 \hat{=}$ VDI, $5 \hat{=}$ GDF). We have connected the discrete values of interest by lines, thus improving the legibility of the diagram. Referring to the notation introduced in Section 2.3, `"D"` stands for the unidirectional approach without exploitation of sparsity (**DM**) and `"N"` represents the better choice out of the one-sided methods due to Newsam and Ramsdell (**NR**). `"F"` and `"B"` denote the forward and backward vertex elimination modes respectively and `"X"` labels the operations counts achieved by applying the vertex elimination heuristic **V_LR**.

The application of an optimized elimination sequence results in the lowest operations count in all cases. The total savings can often reach a factor of 10 compared to **NR**. On the other hand, we observe that a significant decrease of the number of multiplications needed for the calculation of the Jacobian can already be achieved by using the pure forward or backward vertex elimination modes. These two methods do not require the solution of a combinatorial optimization problem and are therefore comparably cheap. Through the solution of the shortest path problem in certain subgraphs of the metagraph, the operations count can often be decreased by another factor of 2 or higher for selected problems. However, the computational effort that has to be put into this optimization is often very high. It will pay off only if the calculated information is used a large number of times, such that we do not care about the time required for running the cross-country optimization process.

We have listed the run-times of different optimization methods for solving the general edge elimination problem for a Gaussian data fitting problem ($n = 5$, $p = 330$, $m = 33$) relative to the time required to run the simple forward vertex elimination sequence in the following table:

V_F	**V_B**	**V_LM**	**V_LR**	**V_LF**	**V_RPL**	**BFRA**	**SA_C**
1	1.11	1.71	2.33	1.89	320.23	76.22	4440.73

By decreasing the operations count from 1815 (**NR**) to 264 (all optimization methods were able to do this) we achieved savings by a factor of nearly 7, which is remarkable. The run-times of the different algorithms differ significantly. However, our most consistent vertex elimination heuristic **V_LR** took only about twice as long as **V_F**, which is acceptable. Furthermore, we have that the minimum out of the operations counts of the forward and backward vertex elimination modes varies depending on the problem. We have observed that **V_LR** delivers nearly optimal results in virtually all cases.

7.3 Optimized Derivative Code

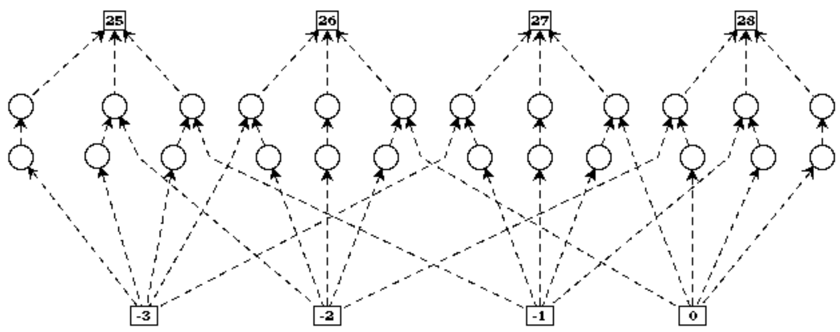

Figure 7.2: Solid Fuel Ignition Problem

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 **V_LR**-based hand-written code (`Code 2`) with a scalar tangent code without exploitation of sparsity (`Code 1`) for the Solid Fuel Ignition problem from the `MINPACK` test problem collection, the source code of which looks as follows:

```
for (i=0;i<ny;i++)
  for (j=0;j<nx;j++) {
    k=nx*i+j;
    ut=0.; ub=0.; ul=0.; ur=0.;
    if (j!=0) ul=x[k-1];
    if (j!=nx-1) ur=x[k+1];
    if (i!=0) ub=x[k-nx];
    if (i!=ny-1) ut=x[k+nx];
    uxx=(-ur+2.*x[k]-ul)*c[0];
    uyy=(-ut+2.*x[k]-ub)*c[1];
    y[k]=uxx+uyy-c[2]*c[3]*exp(x[k]);
  }
```

Figure 7.2 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 derivative codes presented in Appendix B 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`.

7.4 Future Work

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 the theoretical results of this thesis in form of 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.

Appendix A

Example Session

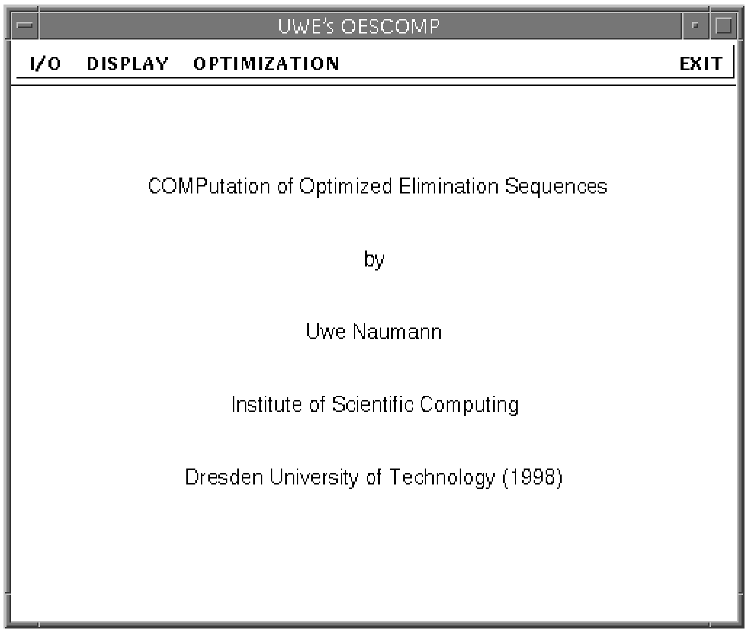

Figure A.1: OESCOMP

With the help of the *Wood function*, taken from the `MINPACK` test problem collection [ACM91], we will run through an example session of **OESCOMP**. This is supposed to give the reader some deeper insight into the usage of our software. Intentionally, we have chosen the example problem to be rather small so we can present the whole range of visualization routines provided by **OESCOMP**. The *Wood function* has the form

$$\mathbf{y} = F(\mathbf{x}), \quad \text{with} \quad F : \mathbb{R}^4 \supseteq D \rightarrow \mathbb{R}^4$$

and it can be written as the following system of non-linear equations:

$$\begin{aligned}
y_0 &= -c_3 x_0 (x_1 - x_0 x_0) - (1 - x_0), \\
y_1 &= c_3 (x_1 - x_0 x_0) + c_4 (x_1 - 1) + c_5 (x_3 - 1), \\
y_2 &= -c_6 x_2 (x_3 - x_2 x_2) - (1 - x_2), \\
y_3 &= c_6 (x_3 - x_2 x_2) + c_4 (x_3 - 1) + c_5 (x_1 - 1).
\end{aligned}$$

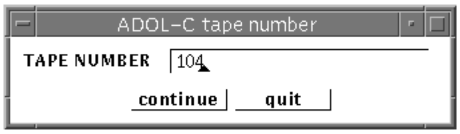

Figure A.2: Load ADOL-C Tape

Here, c_3, c_4, c_5, and c_6 are constants the specific values of which we are not interested in when dealing with the elimination problem in the c-graph.

Before we can start using **OESCOMP** we require the single assignment code of the vector function in form of an ADOL-C (version 1.7) tape. Therefore, we must make some changes to the source code. The result is shown below:

```
#include <stream.h>
#include <iostream.h>

// ADOL-C include files

#include <adouble.h>
#include <adutils.h>

int main() {
  adouble *x, *y, d1, temp1, temp2;
  double *xin, *yout;
  double c3=200., c4=20.2, c5=19.8 , c6=180.;
  int i, tag=104, file=1, n=4;

  xin = new double[n]; yout = new double[n];

  for (i=0;i<n;i++) cin >> xin[i];

  trace_on(tag);
  x = new adouble[n]; y = new adouble[n];

  for (i=0;i<n;i++) x[i]<<=xin[i];

  // begin function source

  d1 = x[0]; temp1 = x[1] - d1 * d1;
  d1 = x[2]; temp2 = x[3] - d1 * d1;
  y[0] = -c3 * x[0] * temp1 - (1.0 - x[0]);
  y[1] = c3 * temp1 + c4 * (x[1] - 1.0) + c5 * (x[3] - 1.0);
  y[2] = -c6 * x[2] * temp2 - (1.0 - x[2]);
  y[3] = c6 * temp2 + c4 * (x[3] - 1.0) + c5 * (x[1] - 1.0);

  // end function source

  for (i=0;i<n;i++) y[i]>>=yout[i];

  delete [] x; delete [] y;
  trace_off(file);

  delete [] xin; delete [] yout;

  return 0;
}
```

A comprehensive introduction into ADOL-C can be found in [GJU96]. One of the key ingredients of AD by overloading as exploited in ADOL-C is the concept of active

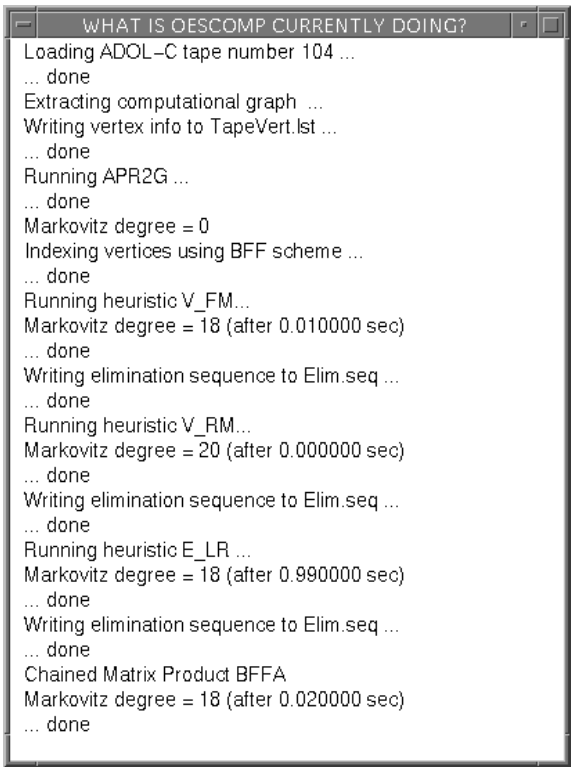

Figure A.3: Status Window

variables. Dealing with the c-graph of a vector function all variables lying on paths that connect any of the independent variables $(x_0, \ldots, x_3)$ with any of the dependent variables $(y_0, \ldots, y_3)$ have to be declared to be of an active type. The scalar active type used in `ADOL-C` carries the name `adouble`, so apart from `x` and `y` the three auxiliary program variables `d1, temp1`, and `temp2` are also declared as `adouble`. Once `ADOL-C` has been installed on the computer the `C++` class `adouble`, its member functions, and the overloaded versions of all arithmetic operations are contained in the file `adouble.c` and its header `adouble.h` which must be included for compilation of the program together with the file `adutils.h`.

`ADOL-C` provides the overloaded operators $\ll=$ and $\gg=$ for specifying independent and dependent variables, respectively. Both actions must be carried out inside the so-called *active section*, marked by the two key words `trace_on()` and `trace_off()`. The argument of the former specifies the number of the *tape* (`tag=104`), whereas the parameter `file` of the latter set to 1 forces the tape to be written to the hard-disc for further processing.

By following the above steps and compiling the code followed by linking it with the `ADOL-C` library `libad.a` we finally end up with a single assignment code of the *Wood function* in form of a *tape* carrying number 104. We are now ready to use **OESCOMP**.

The program is started by typing `oescomp` at the command line. It has a graphic user interface based on the window concept of `LEDA`. Figure A.1 shows the main window which appears at the screen right after starting the program. The first step is to load the *tape* we wish to work with by choosing the corresponding entry in the `I/O` pull-down menu followed by specifying the tape number (Figure A.2).

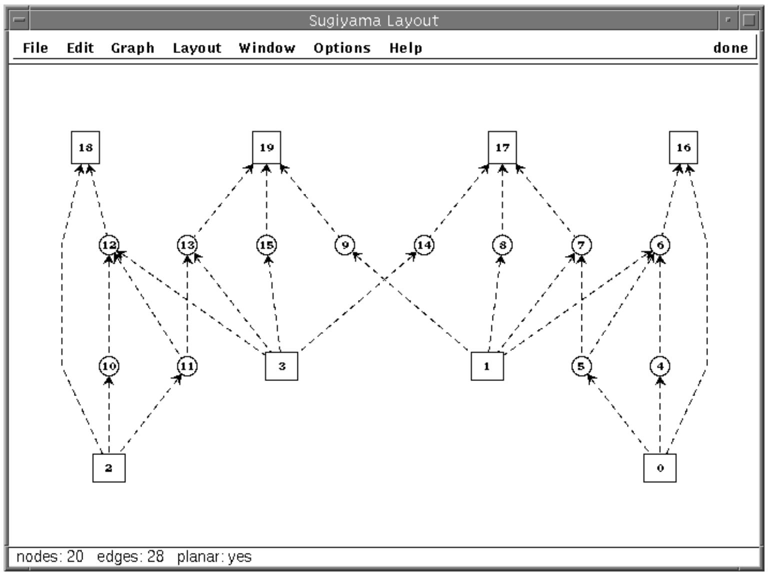

Figure A.4: Computational Graph

A protocol of every action taken while working with **OESCOMP** is written to the *status window* shown in Figure A.3. After the tape has been loaded it is automatically converted into the corresponding c-graph which contains only those vertices that lie on paths connecting the independent with the dependent variables. The user can then decide, which a priori reductions he wants to be performed (`APR2G` $\hat{=}$ deletion of all additive parts such that the generated fill-in is bounded by a small factor), run different heuristics (here forward and backward vertex elimination modes, Lowest-Fill-in-First edge elimination mode), start a simulated annealing process, or optimize the chained local [adjoint] Jacobians product with respect to quotient graphs which are based on different consistent indexing schemes. It is possible to look at certain statistics concerning the c-graph (Figure A.5). Numerous protocols containing, for example, information on all edges or vertices can be written to files. Furthermore, **OESCOMP** enables the user to visualize both the c-graph (Figure A.4) and the quotient graph (Figure A.7) as well as it provides deeper insight into the dynamic programming algorithm for solving the chained matrix problem by allowing to display the table look up graph. Most actions will have to be confirmed by the user (Figure A.6). Within the windows showing the different graphs we can select certain vertices or edges by mouse click and thus visualize the internal data providing information like, for example, vertex type, operations code,

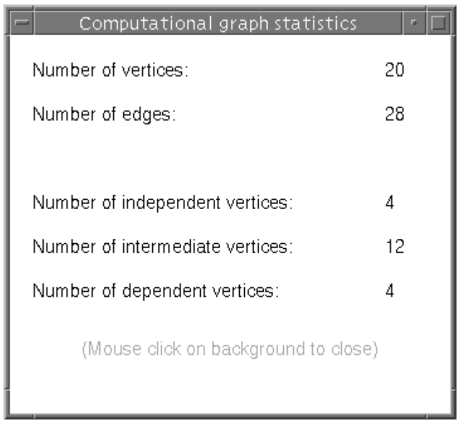

Figure A.5: Graph Statistics

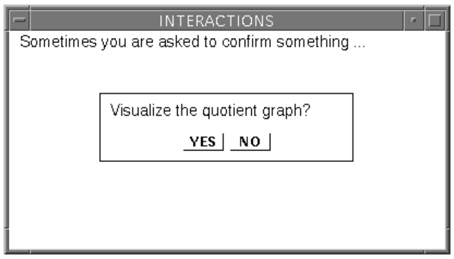

Figure A.6: Confirmation

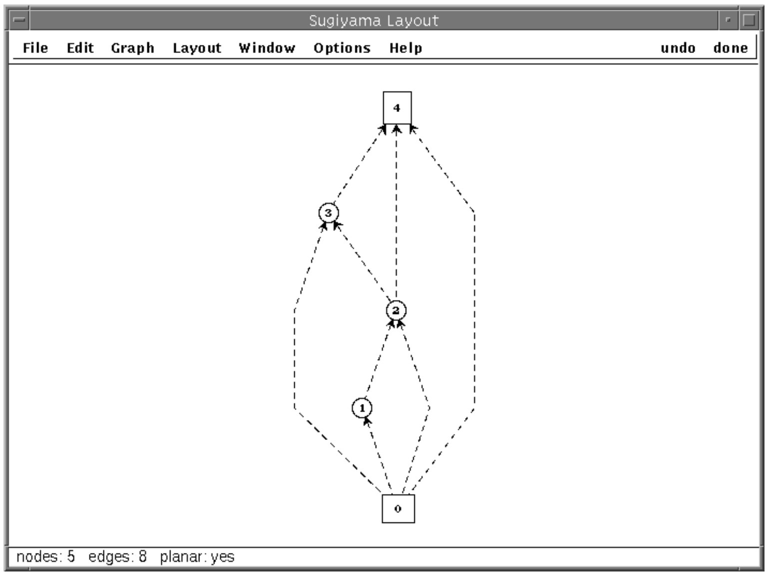

Figure A.7: Quotient Graph

etc. of vertices in the c-graph. An example for this has already been presented in Section 4.4.

The current version of **OESCOMP** should be regarded as a beta-version the functionality of which is subject to extension as outlined in Chapter 7.

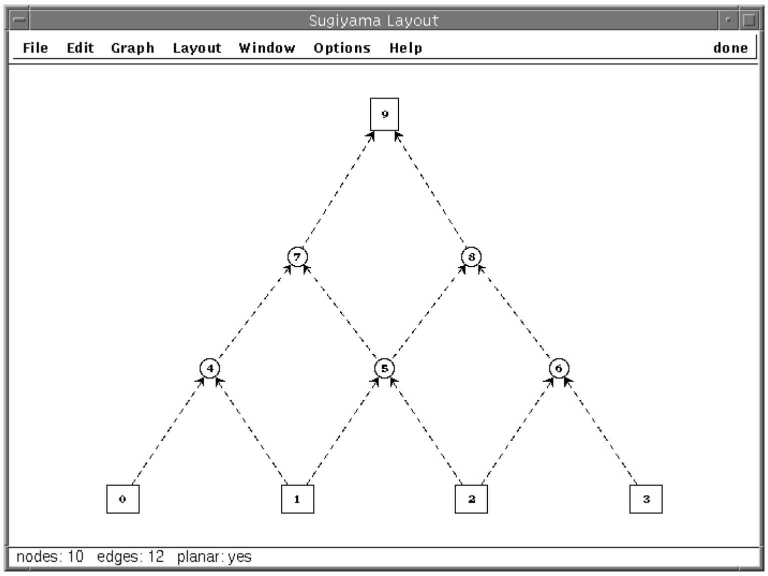

Figure A.8: Table Look Up Graph

Appendix B

Code for Time Measurements

B.1 Code 1

```
int main() {
  double v[29], x[4], jac[4][4], S[4][4], c[4] = {1.,1.,1.,1.};
  int i,j;

  for (i=0;i<4;i++)
    for (j=0;j<4;j++)
      if (i==j) S[i][j]=1.;
      else S[i][j]=0.;
                                                  // start clock
  for (i=0;i<4;i++)
    for (j=0;j<4;j++) jac[i][j]=0.;

  for (i=0;i<4;i++) {
    v[5]=2.*S[i][0]; v[6]=c[0]*(v[5]-S[i][1]);
    v[7]=2.*S[i][0]; v[8]=c[1]*(v[7]-S[i][2]);
    v[9]=exp(x[0])*S[i][0]; v[10]=c[3]*v[9];
    jac[i][0]=v[6]+v[8]-c[2]*v[10];

    v[11]=2.*S[i][1]; v[12]=c[0]*(v[11]-S[i][0]);
    v[13]=2.*S[i][1]; v[14]=c[1]*(v[13]-S[i][3]);
    v[15]=exp(x[1])*S[i][1]; v[16]=c[3]*v[15];
    jac[i][1]=v[12]+v[14]-c[2]*v[16];

    v[17]=2.*S[i][2]; v[18]=c[0]*(v[17]-S[i][3]);
    v[19]=2.*S[i][2]; v[20]=c[1]*(v[19]-S[i][0]);
    v[21]=exp(x[2])*S[i][2]; v[22]=c[3]*v[21];
    jac[i][2]=v[18]+v[20]-c[2]*v[22];

    v[23]=2.*S[i][3]; v[24]=c[0]*(v[23]-S[i][2]);
    v[25]=2.*S[i][3]; v[26]=c[1]*(v[25]-S[i][1]);
    v[27]=exp(x[3])*S[i][3]; v[28]=c[3]*v[27];
    jac[i][3]=v[24]+v[26]-c[2]*v[28]; }
}                                                 // stop clock
```

B.2 Code 2

```
  ...
                                            // start clock
  for (i=0;i<4;i++)
    for (j=0;j<4;j++) jac[i][j]=0.;

  h[0]=2.*c[0]; h[1]=2.*c[1];
  h[2]=c[3]*exp(x[0]); h[3]=c[2]*h[2];

  h[4]=2.*c[0]; h[5]=2.*c[1];
  h[6]=c[3]*exp(x[1]); h[7]=c[2]*h[6];

  h[8]=2.*c[0]; h[9]=2.*c[1];
  h[10]=c[3]*exp(x[2]); h[11]=c[2]*h[10];

  h[12]=2.*c[0]; h[13]=2.*c[1];
  h[14]=c[3]*exp(x[3]); h[15]=c[2]*h[14];

  jac[0][0]=h[0]+h[1]-h[3];
  jac[0][1]=-c[0];
  jac[0][2]=-c[1];

  jac[1][0]=-c[0];
  jac[1][1]=h[4]+h[5]-h[7];
  jac[1][3]=-c[1];

  jac[2][0]=-c[1];
  jac[2][2]=h[8]+h[9]-h[11];
  jac[2][3]=-c[0];

  jac[3][1]=-c[1];
  jac[3][2]=-c[0];
  jac[3][3]=h[12]+h[13]-h[15];
  ...                                       // stop clock
}
```

B.3 Code 3

```
  ...
                                            // start clock
  for (i=0;i<4;i++)
    for (j=0;j<4;j++)
      jac[i][j]=0.;

  h[1]=2.*(c[1]+c[0]);

  h[2]=c[3]*exp(x[0]);
  h[3]=c[2]*h[2];
```

```
h[6]=c[3]*exp(x[1]);
h[7]=c[2]*h[6];

h[10]=c[3]*exp(x[2]);
h[11]=c[2]*h[10];

h[14]=c[3]*exp(x[3]);
h[15]=c[2]*h[14];

jac[0][0]=h[1]-h[3];
jac[0][1]=-c[0];
jac[0][2]=-c[1];

jac[1][0]=-c[0];
jac[1][1]=h[1]-h[7];
jac[1][3]=-c[1];

jac[2][0]=-c[1];
jac[2][2]=h[1]-h[11];
jac[2][3]=-c[0];

jac[3][1]=-c[1];
jac[3][2]=-c[0];
jac[3][3]=h[1]-h[15];
                                             // stop clock
...
```

B.4 Code 4

```
...
                                             // start clock
for (i=0;i<4;i++)
  for (j=0;j<4;j++)
    jac[i][j]=0.;

for (i=0;i<2;i++)
  for (j=0;j<2;j++) {
    k=2*i+j;
    if (j!=0) jac[k][k-1] = -c[0];
    if (j!=1) jac[k][k+1] = -c[0];
    if (i!=0) jac[k][k-2] = -c[1];
    if (i!=1) jac[k][k+2] = -c[1];
    jac[k][k]=2.*(c[0]+c[1]) - c[2]*c[3]*exp(x[k]); }

...                                          // stop clock
```

Appendix C

Documented Source Code (Excerpt)

C.1 OESCOMP.C

1. **OESCOMP** is the implementation of many theoretical results of our research in form of a program for the **COM**putation of **O**ptimized **E**limination **S**equences for intermediate vertices and/or edges in c-graphs. The main part of the software package consists of more than 3000 lines of `C++` code. The size of the complete package involving modules for virtually all approaches to the elimination problem that were described in the thesis adds up to more than 10.000 lines of code. Obviously, it does not make sense to list the entire source code. This is why we have decided to concentrate on one module which approaches the elimination problem by solving the chained local Jacobians multiplication problem using a dynamic programming algorithm as described in Chapter 4. We will explain the implementation details by presenting the relevant excerpts from the different `CWEB`-files the software package consists of starting with the main program.

```
#include <LEDA/graph.h>
#include <LEDA/window.h>
#include <LEDA/graphwin.h>
#include <LEDA/graph_alg.h>
#include "SECG.h"
#include "TLG.h"
#include "QG.h"
  ⟨ Declaration of global variables 2 ⟩
  ⟨ Main program 8 ⟩
```

2. We distinguish between three types of `LEDA`-graphs: The c-graph $*Cg$ is the result of reading the `ADOL-C`-tape and converting it into a directed acyclic graph by removing all obsolete information which we do not need for solving the elimination problem. $*qg$ represents the computational quotient graph which we will build our chained local Jacobians multiplication problem on. Finally, the table lookup graph $*tlgraph$ serves as a basis for the dynamic programming algorithm. We have declared pointers to objects of the corresponding data type which will be assigned to the real data structures below.

OESCOMP comes with an `X-Windows`-based graphical user interface implemented with the help of the appropriate data types provided by `LEDA`. Among others we have two windows one of them (mw) keeping the user informed about the current status of **OESCOMP** and the other ($mw1$) for interactions with the user, like confirming certain actions etc. Refer to the example session in Chapter A of the appendix in order to see what they look like.

⟨ Declaration of global variables 2 ⟩ ≡
```
SECGraph * Cg;
qgraph * qg;
tlg * tlgraph;
window mw(400, 900, "What␣is␣the program␣currently␣doing?");
window mw1 (400, 200, "Interactions");
```
This code is used in section 1.

3. We extract the quotient graph building on a breadth-first-forward numbering of the vertices in the underlying c-graph followed by initializing the local Jacobians associated with each of the super-nodes corresponding to the tangent mode.

⟨ Extract quotient graph 3 ⟩ ≡
```
qg→qgextract(*Cg, 1);
qg→Ctinit(*Cg);
```
This code is used in section 4.

4. Between the allocation of the c-graph $*Cg$ and its conversion into the corresponding quotient graph lie numerous steps like reading the `ADOL-C` tape and running some a priori reductions the source code of which is not presented here.

Following the extraction of the quotient graph we prepare the table lookup graph for the optimization of the *tangent chain* by dynamic programming. This is indicated by passing the value 1 as a parameter into the constructor of an instance of the table lookup graph.

⟨ Create new graphs 4 ⟩ ≡
```
Cg = new SECGraph;
qg = new qgraph ((Cg→Cg).number_of_nodes());
⟨ Extract quotient graph 3 ⟩
tlgraph = new tlg (*qg, 1);
```
This code is used in section 8.

5. Building on an algorithm by Sugiyama which is part of the `AGD`-library we have implemented a routine for visualizing directed acyclic graphs ($dagdraw$). Here the user is asked to confirm a possible visualization by using the mouse in the corresponding window ($mw1$). If the answer is positive $dagdraw$ will open a new window of displaying the `LEDA`-graph component g of the quotient graph $*qg$.

⟨ Visualize quotient graph? 5 ⟩ ≡
```
if (mw1.confirm(string("Visualize␣QG?␣\n"))) dagdraw(qg→g);
```
This code is used in section 8.

6. The dynamic programming algorithm is run in tangent mode (indicated by the 1 as parameter of *optimize*(...)) and the result in form of the minimal number of multiplications required for multiplying the local extended Jacobians together appears in window *mw*.

⟨ Run dynamic programming algorithm 6 ⟩ ≡

```
mw.message(string("Markovitz␣degree␣=␣%d␣\n", tlgraph→optimize(1)));
```

This code is used in section 8.

7. The visualization of the table lookup graph works analogous to the process for displaying the quotient graph.

⟨ Visualize table lookup graph? 7 ⟩ ≡

```
if (mw1.confirm(string("Visualize␣TLG?␣\n"))) dagdraw(tlgraph→g);
```

This code is used in section 8.

8. This brief excerpt from the main part of **OESCOMP** shows the frame for approaching the elimination problem in c-graphs by solving a chained matrix problem by dynamic programming. The dynamically allocated data structures involved in this process are deallocated afterwards.

⟨ Main program 8 ⟩ ≡

```
int main(void)
{
  ⟨ Create new graphs 4 ⟩
  ⟨ Visualize quotient graph? 5 ⟩
  ⟨ Run dynamic programming algorithm 6 ⟩
  ⟨ Visualize table lookup graph? 7 ⟩
  delete Cg;
  delete qg;
  delete tlgraph;
  return (0);
}
```

This code is used in section 1.

C.2 TLG.h

1. The file `TLG.h` contains the definition of the type *tlg*. This structure represents the **table lookup graph** T which serves as the basis for our dynamic programming algorithm. The latter is to solve the chained matrix multiplication problem for minimizing the overall number of multiplications involved in the accumulation of the complete Jacobian of a given vector function.

The outer frame consists of some preprocessor statements for including the required data types and place holders for the definitions of the table lookup graph itself, its nodes and its edges. Neither the nodes nor the edges have any special features. This is why we have decided to omit the corresponding code. We build on the `LEDA` template `GRAPH<...>` defined in `graph.h`. The scalar c-graph CG which contains all the information required to set up the chained matrix optimization problem is defined in `SECG.h`. In order to minimize the run-time of the dynamic programming algorithm we apply it to a **quotient graph** representation of CG the data type declaration of which is provided by `QG.h`.

```
#ifndef TLGH
#define TLGH
#include <LEDA/graph.h>
#include "SECG.h"
#include "QG.h"
  ⟨ tlgnode declaration 11 ⟩
  ⟨ tlg declaration 2 ⟩
#endif
```

2. We define *tlg* as a structure consisting of several member variables, a constructor, a destructor and some member functions for performing certain operations on the class variables.

```
⟨ tlg declaration 2 ⟩ ≡
  struct tlg {
     ⟨ tlg variables 9 ⟩
     ⟨ tlg constructor 7 ⟩
     ⟨ tlg destructor 8 ⟩
     ⟨ tlg member functions 10 ⟩
  };
```

This code is used in section 1.

3. The node matrix *idx* is used for accessing the nodes in *tlg* through a two dimensional index which is crucial for both the forward sweep computing the minimal cost of multiplying the local extended Jacobians together and for the extraction of the optimal parenthesization by a reverse sweep through the table lookup graph.

```
⟨ Allocate tlg node pointer array 3 ⟩ ≡
  idx = new node * [factors];
  for (i = 0; i < factors; i++) idx[i] = new node [factors − i];
  for (i = 0; i < factors; i++)
     for (j = 0; j < factors − i; j++) idx[i][j] = Λ;
```

This code is used in section 7.

4. Level zero of the table lookup graph of the tangent chained matrix product optimization problem contains a node for each vertex in the underlying quotient graph except for the one representing the independent variables since there is no local extended Jacobian associated with it. The factors C of the matrix chain are initialized with the corresponding local extended Jacobians Ct which are associated with each vertex in the quotient graph and contain the local partial derivatives of all vertices with indices in $[lb, ub]$ with respect to their predecessors in the original c-graph.

$\langle$ Initialize *tlg* for tangent chain optimization 4 $\rangle \equiv$
forall_nodes $(n, qg.g)$ {
  if (d) {
    $idx[0][k] = g.new\_node(\,);$
    $g[idx[0][k]].init(dim, factors - f);$
    for $(i = 0;\ i < dim;\ i{+}{+})$
      for $(j = 0;\ j < dim;\ j{+}{+})$ $g[idx[0][k]].C[i][j] = qg.g[n].Ct[i][j];$
    $g[idx[0][k]].lb = qg.g[n].lb;$
    $g[idx[0][k]].ub = qg.g[n].ub;$
    $k{+}{+};$
  }
  $d = 1;$
}

This code is used in section 7.

5. Analogous to the tangent chain, level zero of the table lookup graph of the adjoint chained matrix product optimization problem contains a node for each vertex in the underlying quotient graph except for the one representing the dependent variables as there is no local extended adjoint Jacobian associated with it. Here the factors C are the local extended adjoint Jacobians from the quotient graph.

$\langle$ Initialize *tlg* for adjoint chain optimization 5 $\rangle \equiv$
forall_nodes $(n, qg.g)$ {
  if $(k < factors)$ {
    $idx[0][k] = g.new\_node(\,);$
    $g[idx[0][k]].init(dim, factors - f);$
    for $(i = 0;\ i < dim;\ i{+}{+})$
      for $(j = 0;\ j < dim;\ j{+}{+})$ $g[idx[0][k]].C[i][j] = qg.g[n].Ca[i][j];$
    $g[idx[0][k]].lb = qg.g[n].lb;$
    $g[idx[0][k]].ub = qg.g[n].ub;$
    $k{+}{+};$
  }
}

This code is used in section 7.

6. Building on level zero the entire table lookup graph is allocated by inserting the remaining nodes and connecting them by the corresponding edges. The memory space required for the local extended Jacobians which are associated with each node is reserved a priori by calling the *init*(...) routine.

⟨ Initialize remaining levels of *tlg* 6 ⟩ ≡

```
while (f > 0) {
   s = factors − f;
   for (i = 0; i < f; i++) {
      idx[s][i] = g.new_node();
      g[idx[s][i]].init(dim, s);
      e = g.new_edge(idx[s − 1][i], idx[s][i]);
      e = g.new_edge(idx[s − 1][i + 1], idx[s][i]);
      g[idx[s][i]].lb = g[idx[s − 1][i]].lb;
      g[idx[s][i]].ub = g[idx[s − 1][i + 1]].ub;
   }
   f −−;
}
```

This code is used in section 7.

7. The structure *tlg* has a single constructor which is called with two arguments. *qg* is a quotient graph representation of the c-graph and the *mode* flag indicates whether one intents to optimize the tangent or the adjoint chain. The entire data structure *tlg* is allocated a priori. If the available memory space should not be sufficient for solving a given chained matrix multiplication problem we will recognize it at this initial stage of the dynamic programming algorithm. Notice that we must keep the complete table lookup graph throughout the entire algorithm as the information from all previous levels is required for solving a new subproblem represented by some intermediate node in *tlg*.

⟨ *tlg* constructor 7 ⟩ ≡

```
tlg(qgraph & qg, int mode) : factors(qg.g.number_of_nodes() − 1), dim(qg.dim)
{
   node n;
   edge e;
   int i, j;
   int k = 0, d = 0, f = factors, s;
   ⟨ Allocate tlg node pointer array 3 ⟩
   if (mode ≡ 1) {
      ⟨ Initialize tlg for tangent chain optimization 4 ⟩
   }
   else {
      ⟨ Initialize tlg for adjoint chain optimization 5 ⟩
   }
   f −−;
   ⟨ Initialize remaining levels of tlg 6 ⟩
};
```

This code is used in section 2.

8. The destructor releases the memory space occupied by the node matrix *idx*.

⟨ *tlg* destructor 8 ⟩ ≡

```
inline ~tlg()
{
  for (int i = 0; i < factors; i++) delete [] idx[i];
  delete [] idx;
};
```

This code is used in section 2.

9. The data structure *tlg* contains the following variables: *factors*: number of factors in the chained matrix product; *dim*: number of vertices in the original c-graph (dimension of the square extended local [adjoint] Jacobian); *idx*: matrix of references to all nodes in *tlg* enabeling us to access each node by specifying its two indeces; *g*: the table lookup graph itself consisting of nodes of type *tlgnode* and edges of type *tlgedge*;

⟨ *tlg* variables 9 ⟩ ≡

```
int factors;
int dim;
node **idx;
GRAPH < tlgnode, tlgedge > g;
```

This code is used in section 2.

10. The first two member functions are used internally for checking whether the elimination of some edge results in the elimination of a whole vertex that is incident with it in the quotient graph. The dynamic programming algorithm is run by calling the function *optimize*(...). Further information on all member functions is given in the corresponding definition file `TLG.C`.

⟨ *tlg* member functions 10 ⟩ ≡

```
int vertexelimtan(int, int, int, int);
int vertexelimadj(int, int, int, int);
int optimize(int);
```

This code is used in section 2.

11. Apart from the usual parts like member variables, constructor, destructor a node in *tlg* has to be initialized by a special initialization routine. Furthermore, our program **OESCOMP** allows the visualization of the table lookup graph. Therefore it requires specific *iostream* operators defining the information which can be accessed and the way how to do it.

⟨ *tlgnode* declaration 11 ⟩ ≡

```
struct tlgnode {
    ⟨ tlgnode variables 12 ⟩
    ⟨ tlgnode constructor 13 ⟩
    ⟨ tlgnode initialization routine 14 ⟩
    ⟨ tlgnode destructor 15 ⟩
    ⟨ tlgnode iostream operators 16 ⟩
};
```

This code is used in section 1.

12. A node in the table lookup graph contains the following information: *nn*: dimension of the problem (number of vertices in the original c-graph); *lb*: lower index bound of elemental vertices contained within the current super-node; *ub*: corresponding upper bound; *stage*: stage in the table lookup graph; *cost*: minimal cost to get to this *tlgnode*; *pred*: predecessor associated with this minimal cost; $C[row][column]$: local extended [adjoint] Jacobian (of type `char` in order to minimize the required memory space);

⟨ *tlgnode* variables 12 ⟩ ≡

```
int nn;
int lb, ub, stage, cost;
int pred;
char **C;
```

This code is used in section 11.

13.

⟨ *tlgnode* constructor 13 ⟩ ≡

```
inline tlgnode() : lb(-1), ub(-1), cost(0), pred(0) { };
```

This code is used in section 11.

14. The initialization of a given node in the table lookup graph involves the allocation of the local extended Jacobian associated with it and the setting of both *stage* and *nn*.

⟨ *tlgnode* initialization routine 14 ⟩ ≡

```
tlgnode::init(int n, int st)
{
  stage = st;
  nn = n;
  C = new char * [n];
  for (int i = 0; i < n; i++) {
    C[i] = new char [n];
    for (int j = 0; j < n; j++) C[i][j] = 48;
  }
};
```

This code is used in section 11.

15. The *tlgnode* destructor releases the memory space occupied by the local extended Jacobian.

⟨ *tlgnode* destructor 15 ⟩ ≡

```
inline ~tlgnode()
{
  for (int i = 0; i < nn; i++) delete [] C[i];
  delete [] C;
};
```

This code is used in section 11.

16. The *tlgnode* ostream operator provides read access to the internal information associated with each node. This feature is exploited in the visualization of the table lookup graph which is part of the program **OESCOMP**. We have omitted the code for the corresponding *istream* operator.

⟨ *tlgnode* iostream operators 16 ⟩ ≡

```
friend ostream &operator<<(ostream & o, const tlgnode &n)
{
  o << n.stage << ", [" << n.lb << "," << n.ub << "], " << n.cost << " " << n.pred << "\n";
  if (n.C ≠ Λ)
    for (int i = 0; i < n.nn; i++) {
      for (int j = 0; j < n.nn; j++) o << n.C[i][j];
      o << "\n";
    }
  return o;
}
```

This code is used in section 11.

C.3 TLG.C

1. The file `TLG.C` contains the specifications of all member functions of the structure *tlg* declared in `TLG.h`. It consists of five different first level sections the names of which are listed below.

```
#define TLGC
#include "TLG.h"
  ⟨ Operations count of logical (n × n)-matrix multiplication 2 ⟩
  ⟨ Opcount and sparsity pattern of logical (n × n)-matrix multiplication 3 ⟩
  ⟨ Do we have vertex elimination in tangent mode? 6 ⟩
  ⟨ Do we have vertex elimination in adjoint mode? 9 ⟩
  ⟨ Forward propagation of the cost 13 ⟩
```

2. The first version of *mult*(...) takes two $(n \times n)$-matrices $c1$ and $c2$ and returns the number of scalar multiplications required for the computation of the matrix product $c2 \cdot c1$. Representing local extended [adjoint] Jacobians both $c1$ and $c2$ are square lower triangular matrices $\Rightarrow$ $i2 < i1$ and $i3 < i2$ in the **for**-loops.

```
⟨ Operations count of logical (n × n)-matrix multiplication 2 ⟩ ≡
  int mult(char **c1, char **c2, int n)
  {
    int mults = 0;
    for (int i1 = 0; i1 < n; i1++)
      for (int i2 = 0; i2 < i1; i2++)
        for (int i3 = 0; i3 < i2; i3++) mults += (c2[i1][i2] & 1) ∧ (c1[i2][i3] & 1);
    return mults;
  }
```

This code is used in section 1.

3. Apart from calculating the number of scalar multiplications involved in $c2 \cdot c1$ a second, overloaded, instance of *mult(...)* computes the sparsity pattern of the result of multiplying $c2$ and $c1$ explicitely and returns it as C.

```
⟨ Opcount and sparsity pattern of logical (n × n)-matrix multiplication 3 ⟩ ≡
  int mult(char **C, char **c1, char **c2, int n)
  {
    int mults = 0, m;
    for (int i1 = 0; i1 < n; i1++)
      for (int i2 = 0; i2 < i1; i2++)
        for (int i3 = 0; i3 < i2; i3++) {
          m = (c2[i1][i2] & 1) ∧ (c1[i2][i3] & 1);
          if (m) {
            C[i1][i3] = 49;
            mults += m;
          }
        }
    return mults;
  }
```

This code is used in section 1.

4. If in addition to $[v_{ub+1}]$ there are other successors of $[v_{lb}]$ in the quotient graph then there must be at least one elemental vertex in the initial quotient graph represented by stage zero in the table lookup graph which has successors different from the ones contained within $[v_{ub+1}]$.

$\langle$ Are there other successors? 4 $\rangle \equiv$

```
for (l = 0; l < factors; l++) {
  for (j = ub + 1; j < dim; j++)
    if ((g[idx[0][l]].C[j][k] & 1) ∧ ((j > g[idx[s − 1][i + 1]].ub) ∨ (j < g[idx[s − 1][i + 1]].lb))) {
      succ = 1;
      exit;
    }
  if (succ) exit;
}
```

This code is used in section 6.

5. In order to delete $[v_{lb}]$ from the quotient graph one simply has to delete all its in-edges as its out-degree is already zero. This is done by setting the entries in the k-th row ($lb \le k \le ub$) in the corresponding local extended Jacobian equal to 0 (ascii: $'0' \hat{=} 48$).

$\langle$ Delete all in-edges 5 $\rangle \equiv$

```
for (l = 0; l < dim; l++) g[idx[s][i]].C[k][l] = 48;
```

This code is used in section 6.

6. *vertexelimtan(...)* checks whether the super-node $[v_{lb}] = [v_{ub}]$ represented by the i-th vertex at the $(s-1)$-st stage in the table lookup graph ($g[idx[s-1][i]] \in T$) should disappear as a result of the forward elimination of the super-edge connecting it with its successor $[v_{ub+1}]$ in the *quotient graph* $CG/\sim$ represented by $g[idx[s-1][i+1]]$ in T. Obviously, this is the case only if the eliminated super-edge $([ub],[ub+1])$ was the only out-edge of $[v_{lb}]$.

$\langle$ Do we have vertex elimination in tangent mode? 6 $\rangle \equiv$

```
int tlg::vertexelimtan(int s, int i, int lb, int ub)
{
  int k, j, l, succ;
  for (k = lb; k ≤ ub; k++) {
    succ = 0;
    ⟨ Are there other successors? 4 ⟩
    if (¬succ) ⟨ Delete all in-edges 5 ⟩
  }
  return succ;
}
```

This code is used in section 1.

7. A principle equivalent to the one used in the tangent mode is applied.

$\langle$ Are there other predecessors? 7 $\rangle \equiv$

```
for (l = 0; l < factors; l++) {
  for (j = 0; j < lb; j++)
    if ((g[idx[0][l]].C[k][j] & 1) ∧ ((j > g[idx[s − 1][i]].ub) ∨ (j < g[idx[s − 1][i]].lb))) {
      pred = 1;
      exit;
    }
  if (pred) exit;
}
```

This code is used in section 9.

8. The removal of out-edges corresponds to setting the entries in the k-th column ($lb \leq k \leq ub$) in the local extended adjoint Jacobian to 0.

$\langle$ Delete all out-edges 8 $\rangle \equiv$
  for $(l = 0;\ l < dim;\ l{+}{+})\ g[idx[s][i]].C[l][k] = 48;$

This code is used in section 9.

9. *vertexelimadj(...)* checks the possibility of $[v_{lb}] = [v_{ub}]$ disappearing as a result of the reverse elimination of $([lb-1],[lb])$. Again, this is the case only if the cardinality of $P_{[lb]}$ before the elimination of $([lb-1],[lb])$ was one.

$\langle$ Do we have vertex elimination in adjoint mode? 9 $\rangle \equiv$
  int *tlg* :: *vertexelimadj* (**int** s, **int** i, **int** lb, **int** ub)
  {
    int $k,\ j,\ l,\ pred;$
    for $(k = lb;\ k \leq ub;\ k{+}{+})$ {
      $pred = 0;$
      $\langle$ Are there other predecessors? 7 $\rangle$
      if $(\neg pred)$ $\langle$ Delete all out-edges 8 $\rangle$
    }
    return $pred;$
  }

This code is used in section 1.

10. The result C of the matrix product $C_2 \cdot C_1$ will contain all non-zero entries of either one of the factors except for those which represent edges leading from vertices with indeces in $[l_1, u_1]$ to vertices indeces of which are elements of $[l_2, u_2]$. C will not have any non-zero entries in columns $l_1 \cdots u_1$. The possible removal of $[v_{l_2}] \in CG/\sim$ is examined by *vertexelimtan(...)*.

$\langle$ Initialize new local Jacobian in tangent mode 10 $\rangle \equiv$
  for $(i1 = 0;\ i1 < dim;\ i1{+}{+})$
    for $(i2 = 0;\ i2 < dim;\ i2{+}{+})$
      if $(((g[idx[s-1][i]].C[i1][i2] \,\&\, 1) \vee (g[idx[s-1][i+1]].C[i1][i2] \,\&\, 1)) \wedge ((i2 > g[idx[s-1][i]].ub) \vee (i2 < g[idx[s-1][i]].lb)))\ g[idx[s][i]].C[i1][i2] = 49;$
  $vertexelimtan(s, i, g[idx[s-1][i]].lb, g[idx[s-1][i]].ub);$

This code is used in section 13.

11. The result $\bar{C}$ of the matrix product $\bar{C}_2 \cdot \bar{C}_1$ will contain all non-zero entries of either one of the factors except for those which represent edges leading from vertices with indeces in $[l_1, u_1]$ to vertices indeces of which are elements of $[l_2, u_2]$. $\bar{C}$ will not have any non-zero entries in rows $l_2 \cdots u_2$. *vertexelimadj(...)* checks the possible removal of $[v_{l_1}] \in CG/\sim$.

$\langle$ Initialize new local Jacobian in adjoint mode 11 $\rangle \equiv$
  for $(i1 = 0;\ i1 < dim;\ i1{+}{+})$
    for $(i2 = 0;\ i2 < dim;\ i2{+}{+})$
      if $(((g[idx[s-1][i]].C[i1][i2] \,\&\, 1) \vee (g[idx[s-1][i+1]].C[i1][i2] \,\&\, 1)) \wedge ((i1 > g[idx[s-1][i+1]].ub) \vee (i1 < g[idx[s-1][i+1]].lb)))\ g[idx[s][i]].C[i1][i2] = 49;$
  $vertexelimadj(s, i, g[idx[s-1][i+1]].lb, g[idx[s-1][i+1]].ub);$

This code is used in section 13.

12. By exploiting the *optimal substructure* principle of dynamic programming we get for our chained matrix problem

$$m_{ji} = \begin{cases} 0, & \text{if} \quad i = j, \\ \mathrm{MIN}_{i \le k < j}(m_{ki} + m_{j,k+1} + p_{[j,k+1][ki]}), & \text{if} \quad \mathrm{j > i} \end{cases}$$

where $p_{[j,k+1][ki]}$ is the cost of multiplying $C_{[j,k+1]}$ by $C_{[ki]}$. For each vertex in the table lookup graph we thus compute the minimal cost of getting there (min) and the index ($minidx$) indicating the way in which it can be split into the optimal solution of two subproblems.

```
⟨ Run local minimization 12 ⟩ ≡
  for (k = 0; k < s; k++) {
    if (k ≡ 0) cost = mult(g[idx[s][i]].C, g[idx[k][i]].C, g[idx[s − 1 − k][i + 1 + k]].C, dim);
    else cost = mult(g[idx[k][i]].C, g[idx[s − 1 − k][i + 1 + k]].C, dim);
    cost += g[idx[k][i]].cost + g[idx[s − 1 − k][i + 1 + k]].cost;
    if (cost < min) {
      min = cost;
      minidx = k;
    }
    g[idx[s][i]].cost = min;
    g[idx[s][i]].pred = minidx;
  }
```

This code is used in section 13.

13. In order to solve the chained matrix multiplication problem we iterate over all stages (s) in the table lookup graph. Within each stage we consider all nodes (i) each of them representing the optimal solution of a particular subproblem. Finally, *optimize(...)* returns the minimal number of scalar multplications required for computing the chained matrix product. The optimal parenthesization can be extracted using the value of the *pred* field of each single node in the table lookup graph.

```
⟨ Forward propagation of the cost 13 ⟩ ≡
  int tlg::optimize(int mode)
  {
    int i, i1, i2, k, s, cost, min, minidx;
    for (s = 1; s < factors; s++)
      for (i = 0; i < factors − s; i++) {
        min = dim ∗ dim ∗ dim;
        minidx = 0;
        if (mode) {
           ⟨ Initialize new local Jacobian in tangent mode 10 ⟩
        }
        else {
           ⟨ Initialize new local Jacobian in adjoint mode 11 ⟩
        }
        ⟨ Run local minimization 12 ⟩
      }
    return min;
  }
```

This code is used in section 1.

C.4 QG.h

1. The file `QG.h` contains the declaration of the **quotient graph** *qg* several instances of which can be derived from a given c-graph. Building on *qg* a dynamic programming algorithm can be used to approach the problem of accumulating the Jacobian of a vector function using as little multiplications as possible by optimizing a chained matrix product the factors of which represent local extended [adjoint] Jacobians of the so-called super-nodes in *qg*.

```
#ifndef QGH
#define QGH
#include <LEDA/graph.h>
#include "SECG.h"
  ⟨ snode declaration 6 ⟩
  ⟨ qgraph declaration 2 ⟩
#endif
```

2.

```
⟨ qgraph declaration 2 ⟩ ≡
  struct qgraph {
    ⟨ qgraph variables 3 ⟩
    ⟨ qgraph constructor 4 ⟩
    ⟨ qgraph member functions 5 ⟩
  };
```

This code is used in section 1.

3. A quotient graph consists of **super-nodes** and **super-edges** each of them representing collections of elemental vertices or edges from the scalar c-graph. In our implementation the super-edges do not carry any additional information so we use the `LEDA` internal edge type. *dim* is the number of elemental vertices in the c-graph.

```
⟨ qgraph variables 3 ⟩ ≡
  GRAPH < snode , edge > g;
  int dim;
```

This code is used in section 2.

4.

```
⟨ qgraph constructor 4 ⟩ ≡
  inline qgraph(int d) : dim(d) { };
```

This code is used in section 2.

5.

```
⟨ qgraph member functions 5 ⟩ ≡
  void qgextract ( SECGraph & , int ) ;
  void Cainit ( SECGraph & ) ;
  void Ctinit ( SECGraph & ) ;
```

This code is used in section 2.

6. The super-nodes are represented by instances of the structure *snode*.

```
⟨ snode declaration 6 ⟩ ≡
  struct snode {
    ⟨ snode variables 7 ⟩
    ⟨ snode constructor 8 ⟩
    ⟨ snode iostream operators 9 ⟩
  };
```

This code is used in section 1.

7. A super-node contains the following information: *nn* : number of vertices in the original c-graph. This number is equal to the dimension of the square matrices representing the local extended [adjoint] Jacobians. *lb*, *ub*: lower and upper bound of indices of the elemental vertices that belong to a super-node; *nl*: list of elemental vertices in a super-node; *Ct* [*Ca*]: local extended [adjoint] Jacobian representing one factor in the chained matrix product;

⟨ *snode* variables 7 ⟩ ≡
```
int nn;
int lb, ub;
list<node> nl;
char **Ct;
char **Ca;
```
This code is used in section 6.

8.

⟨ *snode* constructor 8 ⟩ ≡
```
inline snode() : nn(0)
{
  Ct = Ca = Λ;
  nl.clear();
};
```
This code is used in section 6.

9. The LEDA-based graphical user interface of our program **OESCOMP** allows us to visualize the internal data of every super-node. To do that we require a special *ostream* operator for accessing the values of member variables of the class *gqraph*. This information will be part of the *Node Setup Window* an example of which is shown in Section 4.4.

⟨ *snode* iostream operators 9 ⟩ ≡
```
friend ostream &operator<<(ostream & o, const snode &n)
{
  int i, j;
  o << n.nn << ", [" << n.lb << "," << n.ub << "] \n";
  if (n.Ct ≠ Λ)
    for (i = 0; i < n.nn; i++) {
      for (j = 0; j < n.nn; j++) o << n.Ct[i][j];
      o << " \n";
    }
  if (n.Ca ≠ Λ)
    for (i = 0; i < n.nn; i++) {
      for (j = 0; j < n.nn; j++) o << n.Ca[i][j];
      o << " \n";
    }
  return o;
}
```
This code is used in section 6.

C.5 QG.C

1. This file contains the specifications of all member functions of the data type *qg* which is defined in `QG.h`.

```
#define QGRAPHC
#include "QG.h"
  ⟨ Quotient graph extractor 10 ⟩
  ⟨ Initialize local Jacobian 9 ⟩
  ⟨ Initialize local adjoint Jacobian 7 ⟩
```

2. The mutually unreachable minimal vertices ($typ() \equiv Indep$) representing the independent variables in the c-graph Cg form the only minimal super-node.

```
⟨ Create minimal super-node 2 ⟩ ≡
  n1 = g.new_node();
  g[n1].lb = 0;
  g[n1].nn = Cg.Cg.number_of_nodes();
  forall_nodes (m, Cg.Cg)
    if ((Cg.Cg)[m].typ() ≡ Indep) g[n1].nl.append(m);
  g[n1].ub = Cg.number_of_indep() − 1;
```

This code is used in section 5.

3. Analogous, the mutually unreachable maximal vertices ($typ() \equiv Dep$) representing the dependent variables in Cg are collected in the only maximal super-node.

```
⟨ Create maximal super-node 3 ⟩ ≡
  n2 = g.new_node();
  g[n2].ub = Cg.Cg.number_of_nodes() − 1;
  g[n2].nn = Cg.Cg.number_of_nodes();
  forall_rev_nodes(m, Cg.Cg)
      if ((Cg.Cg)[m].typ() ≡ Dep) g[n2].nl.append(m);
      g[n2].lb = g[n2].ub − Cg.number_of_dep() + 1;
```

This code is used in section 5.

4. Whenever there is an edge in Cg which connects a member of one super-node with a member of another super-node then these two are connected by a super-edge in $CG/\sim$.

```
⟨ Create super-edges 4 ⟩ ≡
  forall_nodes (n, g)
    forall_nodes (m, g) {
      c = 0;
      forall (n1, g[n].nl) {
        forall (n2, g[m].nl) {
          forall_in_edges (e, n2)
            if ((Cg.Cg).source(e) ≡ n1) {
              c = 1;
              qe = g.new_edge(n, m);
              break;
            }
          if (c) break;
        }
        if (c) break;
      }
    }
```

This code is used in section 5.

5. Building on the original c-graph Cg one successively extracts the super-nodes and connects them by the corresponding super-edges.

$\langle$ Quotient graph extraction 5 $\rangle \equiv$

```
qn.clear();
⟨Create minimal super-node 2⟩
i = Cg.number_of_indep() − 1;
forall_nodes (n, Cg.Cg) {
  if (((Cg.Cg)[n].typ() ≠ Dep) ∧ ((Cg.Cg)[n].typ() ≠ Indep)) {
    i++;
    if (qn.empty()) {
      m = g.new_node();
      g[m].lb = i;
      g[m].ub = i;
      g[m].nn = Cg.Cg.number_of_nodes();
      g[m].nl.append(n);
      qn.append(n);
    }
    else {
      c = 0;
      forall (n1, qn) {
        forall_out_edges (e, n1)
          if ((Cg.Cg).target(e) ≡ n) {
            qn.clear();
            m = g.new_node();
            g[m].lb = i;
            g[m].ub = i;
            g[m].nn = Cg.Cg.number_of_nodes();
            g[m].nl.append(n);
            qn.append(n);
            c = 1;
            break;
          }
        if (c) break;
      }
      if (¬c) {
        g[m].nl.append(n);
        qn.append(n);
        g[m].ub++;
      }
    }
  }
}
⟨Create maximal super-node 3⟩
⟨Create super-edges 4⟩
```

This code is used in section 10.

6. The local extended adjoint Jacobian Ca of the current super-node is allocated and initialized to the zero matrix.

⟨ Allocate local adjoint Jacobian 6 ⟩ ≡

```
g[qn].Ca = new char * [g[qn].nn];
for (i = 0; i < g[qn].nn; i++) g[qn].Ca[i] = new char [g[qn].nn];
for (i = 0; i < g[qn].nn; i++)
  for (j = 0; j < g[qn].nn; j++) g[qn].Ca[i][j] = 48;
```

This code is used in section 7.

7. The local extended adjoint Jacobian of a super-node qn contains a non-zero entry for all edges emanating from elemental vertices contained within qn.

⟨ Initialize local adjoint Jacobian 7 ⟩ ≡

```
void qgraph::Cainit(SECGraph & Cg)
{
  edge e;
  node qn, n, m, n1;
  int i, j, k1, k2;
  forall_nodes (qn, g) {
    ⟨ Allocate local adjoint Jacobian 6 ⟩
    k1 = 0;
    forall_nodes (n, Cg.Cg) {
      k2 = 0;
      forall_nodes (m, Cg.Cg) {
        if (k2 > k1)
          forall (n1, g[qn].nl)
            if (n ≡ n1)
              forall_out_edges (e, n)
                if (Cg.Cg.target(e) ≡ m) g[qn].Ca[k2][k1] = 49;
        k2++;
      }
      k1++;
    }
  }
}
```

This code is used in section 1.

8.

⟨ Allocate local Jacobian 8 ⟩ ≡

```
g[qn].Ct = new char * [g[qn].nn];
for (i = 0; i < g[qn].nn; i++) g[qn].Ct[i] = new char [g[qn].nn];
for (i = 0; i < g[qn].nn; i++)
  for (j = 0; j < g[qn].nn; j++) g[qn].Ct[i][j] = 48;
```

This code is used in section 9.

9. The local extended Jacobian of a super-node *qn* contains a non-zero entry for all edges leading into elemental vertices contained within *qn*.

⟨ Initialize local Jacobian 9 ⟩ ≡

```
void qgraph::Ctinit(SECGraph &Cg)
{
  edge e;
  node qn, n, m, n1;
  int i, j, k1, k2;
  forall_nodes (qn, g) {
    ⟨ Allocate local Jacobian 8 ⟩
    k1 = 0;
    forall_nodes (n, Cg.Cg) {
      k2 = 0;
      forall_nodes (m, Cg.Cg) {
        if (k2 > k1)
          forall (n1, g[qn].nl)
            if (m ≡ n1)
              forall_in_edges (e, m)
                if (Cg.Cg.source(e) ≡ n) g[qn].Ct[k2][k1] = 49;
        k2++;
      }
      k1++;
    }
  }
}
```

This code is used in section 1.

10. The quotient graph can be extracted based on either a breadth-first-forward or a breadth-first-reverse indexing scheme of the elemental vertices in the c-graph. These two schemes represent two special cases among a large variety of consistent numberings. The resulting quotient graphs are in general not equal.

⟨ Quotient graph extractor 10 ⟩ ≡

```
void qgraph::qgextract(SECGraph &Cg, int mode)
{
  list⟨node⟩ qn;
  int i, j;
  int c;
  node n, m, n1, n2;
  edge e, qe;
  if (mode) {
    Cg.BFFsort();
    ⟨ Quotient graph extraction 5 ⟩
  }
  else {
    Cg.BFRsort();
    ⟨ Quotient graph extraction 5 ⟩
  }
}
```

This code is used in section 1.

C.6 SECG.h

1. The class *SECGraph* is a direct offspring of *BCGraph*. It serves as the basis for running all kinds of heuristics for computing nearly optimal elimination sequences as well as both the simulated annealing and the dynamic programming algorithms are built on it.

```
#ifndef SECGRAPHH
#define SECGRAPHH
#include "BCG.h"
  ⟨ Class declaration 2 ⟩
#endif
```

2. *SECGraph* inherits from *BCGraph* making all variables which are declared **public** in *BCGraph* public in *SECGraph* too.

```
⟨ Class declaration 2 ⟩ ≡
  class SECGraph : public BCGraph {
  protected:
     ⟨ Protected class variables 3 ⟩
     ⟨ Protected internal routines 4 ⟩
  public:
     ⟨ Class constructor 5 ⟩
     ⟨ Public interface routines 6 ⟩} ;
```

This code is used in section 1.

3. *markovitz* contains the overall Markovitz degree corresponding to the actual stage in the meta-graph.

```
⟨ Protected class variables 3 ⟩ ≡
  int markovitz;
  ...
```

This code is used in section 2.

4. The protected internal routines are determined for the use within the class *SECGraph* and could also be used by classes which are derived from *SECGraph*. As an example *deletenode*(...) realizes the physical deletion of a vertex together with all edges incident with it from the c-graph.

```
⟨ Protected internal routines 4 ⟩ ≡
  void deletenode (node & , unsigned short);
  ...
```

This code is used in section 2.

5. The class constructor calls the constructor of *BCGraph* and initializes the overall Markovitz degree to zero.

```
⟨ Class constructor 5 ⟩ ≡
  inline SECGraph(void) : BCGraph( ) { markovitz = 0; }
```

This code is used in section 2.

6. There are several interface routines for working with *SECGraph*. They are defined in the file `SECG.C` which we are not going to present here.

Apart from the routine for eliminating a vertex from the c-graph according to the elimination rules we have listed the two vertex indexing algorithms used for the extraction of alternative quotient graphs.

```
⟨ Public interface routines 6 ⟩ ≡
  void eliminatenode(node);
  void BFRsort(void);
  void BFFsort(void);
  ...
```

This code is used in section 2.

C.7 BCG.h

1. The class *BCGraph* is the frame for the internal representation of the c-graph of a vector function. An instance of *BCGraph* will be the result of converting the `ADOL-C` tape (representing the evaluation trace) into a directed acyclic graph. Furthermore, all a priori reduction routines are member functions of this class. Again we will present only the relevant parts of the class declaration.

We start with the inclusion of the declarations of the classes *OpNode* and *OpEdge* and the `LEDA`-graph type. The latter provides a template-based parameterized version of a graph allowing to create instances with vertices of type *OpNode* and edges of type *OpEdge*.

```
#ifndef BCGRAPHH
#define BCGRAPHH
#include "OpNode.h"
#include "OpEdge.h"
#include <LEDA/graph.h>
  ⟨ Class declaration 2 ⟩
#endif
```

2. The high-level version of the declaration of the class *BCGraph* lists all components which will be refined below. The class *SECGraph* which will be defined later is declared to be a **friend** of *BCGraph* thus allowing its member functions to access the protected class variables of *BCGraph*.

```
⟨ Class declaration 2 ⟩ ≡
  class BCGraph {
  private:
     ⟨ Private class routines 3 ⟩
  protected:
     ⟨ Protected class variables 4 ⟩
  public:
     ⟨ Public class variables 5 ⟩
     ⟨ Class constructor 6 ⟩
     ⟨ Copy constructor 7 ⟩
     ⟨ Assignment operator 8 ⟩
     ⟨ Public interface routines 9 ⟩
  friend class SECGraph; } ;
```

This code is used in section 1.

3. There are numerous private member functions mainly for the conversion of the `ADOL-C`-tape into the corresponding graph representation.

```
⟨ Private class routines 3 ⟩ ≡
  void tapeopen(short);
  ...
```

This code is used in section 2.

4. *indep*, *inter* and *dep* are the numbers of vertices representing independent, intermediate and dependent variables in the underlying single assignment code.

```
⟨ Protected class variables 4 ⟩ ≡
  int indep;
  int inter;
  int dep;
  ...
```

This code is used in section 2.

5. The public class variable Cg is a `LEDA` graph with vertices of type *OpNode* and edges of type *OpEdge* which are defined in the files `OpNode.h` and `OpEdge.h` respectively. It represents the actual structure of the c-graph, neither containing any information about the values of the intermediate variables nor having labels representing the local partial derivatives attached to its edges.

⟨ Public class variables 5 ⟩ ≡

```
GRAPH < OpNode, OpEdge > Cg;
```

This code is used in section 2.

6.

⟨ Class constructor 6 ⟩ ≡

```
inline BCGraph(void) : indep(0), inter(0), dep(0) { }
```

This code is used in section 2.

7.

⟨ Copy constructor 7 ⟩ ≡

```
BCGraph(const BCGraph &BCg) : Cg(BCg.Cg), indep(BCg.indep), inter(BCg.inter), dep(BCg.dep)
  { }
```

This code is used in section 2.

8. Sometimes it is necessary to make a precise copy of the c-graph. This can be done by creating a new instance of *BCGraph* and performing an assignment using this overloaded version of the assignment operator.

⟨ Assignment operator 8 ⟩ ≡

```
BCGraph &operator=(const BCGraph &BCg)
{
  node n;
  if (this ≡ &BCg) return *this;
  Cg = BCg.Cg;
  forall_nodes(n, Cg) Cg[n] = (BCg.Cg)[n];
  indep = BCg.indep;
  inter = BCg.inter;
  dep = BCg.dep;
  return *this;
}
```

This code is used in section 2.

9. There are several interface routines for accessing member variables of *BCGraph*. They are defined in the corresponding file `BCG.C` the presentation of the source code of which is not required for the further understanding.

⟨ Public interface routines 9 ⟩ ≡

```
const int &number_of_indep(void);
const int &number_of_intermed(void);
const int &number_of_dep(void);
...
```

This code is used in section 2.

Appendix D

Notation and Software

D.1 Notation

This is a list of selected notations which might be useful for understanding parts of this thesis separately.

F	vector function
$\mathbf{x}$	vector of n independent variables
$\mathbf{y}$	vector of m dependent variables
φ_j	scalar valued elemental function
Φ_j	vector valued elemental function
J	Jacobian matrix of F at the current argument
J_e	extended Jacobian matrix of F at the current argument
c_{ji}	local partial derivative
CG	computational graph of F
CG'	converse computational graph
LG	computational line graph
$CG/\sim$	computational quotient graph
X	set of minimal [independent] vertices
Y	set of maximal [dependent] vertices
Z	intermediate vertices
$\mathcal{I}$	(consistent) numbering of vertices in CG
v_j	j-th vertex in CG
(i, j)	edge with source v_i and target v_j in CG
$(i, \ldots, j)$	path with source v_i and target v_j in CG
C_i	(additive) local extended Jacobian of v_i
B_i	(multiplicative) local extended Jacobian of v_i
C_N	(additive) local extended Jacobian of a set of vertices v_i with $i \in N$
$\bar{C}_i$	(additive) local extended adjoint Jacobian of v_i

$\mathbf{e}_i$	i-th Cartesian basis vector
I	identity matrix
M	metagraph
V_M	set of vertices (stages) in M
$\mathbf{OPS}^k\{v_j\}$	Markovitz degree of v_j at stage k in M
$\mathbf{OPS_F}^k\{(i,j)\}$	Forward Markovitz degree of (i,j) at stage k in M
$\mathbf{OPS_B}^k\{(i,j)\}$	Backward Markovitz degree of (i,j) at stage k in M
ρ	restriction on general edge elimination problem
M_ρ	subgraph of M induced by restriction ρ
$M_\mathbf{V}$	vertex metagraph
$\mathbf{Cost}\{J\}$	number of scalar multiplications involved in the accumulation of the complete Jacobian (length of a path in M)
$\mathbf{Cost}_\rho\{J\}$	length of a path in M_ρ
$\mathbf{OPS}\{\rho\}$	number of scalar multiplications involved in the extraction of M_ρ
$\mathcal{D}_2$	special vertex-edge discrepancy for $\|Z\| = 2$
$\mathcal{D}_p$	general vertex-edge discrepancy for $\|Z\| = p$
DF	dense forward mode of AD
DR	dense reverse mode of AD
DM	"cheaper" choice out of **DF** and **DR** wrt. $\mathbf{Cost}_\rho\{J\}$
NR	cheaper" choice out of one-sided methods by Newsam and Ramsdell wrt. $\mathbf{Cost}_\rho\{\mathbf{J}\}$
V_F	forward vertex elimination mode
V_B	backward vertex elimination mode
$\mathcal{N}$	number of different paths in CG
$\mathcal{L}$	total length of all paths in CG
$\|P_j\|_k$	number of predecessors of v_j in CG at stage k in M
$\|S_j\|_k$	number of successors of v_j in CG at stage k in M
$\underline{\underline{\|P_j\|}}$	minimal number of predecessors of v_j wrt. any stage in M
$\overline{\overline{\|S_j\|}}$	maximal number of successors of v_j wrt. any stage in M
V_$*$	different heuristics for vertex elimination
E_$*$	different heuristics for edge elimination
SA_$*$	different simulated annealing algorithms

We use small letters for scalars and bold small letters for vectors. Matrices and sets are denoted by capital letters. It should not be difficult to distinguish between the two depending on the context.

D.2 Software

The implementation of our theoretical results and all practical experiments were done using SUN Sparc20 workstations with Solaris 2.5. The following software was used:

- `ADOL-C` Library for Automatic Differentiation by Overloading in C, version 1.7
  `http://www.math.tu-dresden.de/wir/project/wwwadolc/index.html`
- `LEDA` Library of Efficient Data Types and Algorithms, version 3.7.1
  `http://www.mpi-sb.mpg.de/LEDA/leda.html`
- `AGD` Library of Algorithms for Graph Drawing, version 0.9 beta
  `http://www.mpi-sb.mpg.de/AGD/`
- `g++` GNU C++ compiler, version 2.8
  `ftp://ftp.gnu.org/pub/gnu`
- `CWEB` System of Structured Documentation, version 3.4
  `ftp://labrea.stanford.edu/pub/cweb/`
- `NEOS` Network Enabled Optimization Server
  `http://www.mcs.anl.gov/otc/Server/neos//homepage.html`
- `MINPACK` Test Problem Collection
  `ftp://ftp.netlib.org/minpack/`
- `f2c` Fortran77 to C/C++ Translator
  `ftp://ftp.netlib.org/f2c/`

Bibliography

[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.

[AGD98] *A library of algorithms for graph drawing*, MPI Saarbrücken, http://www.mpi-sb.mpg.de/AGD/, 1998.

[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.

[BKBC96] C. Bischof, P. Khademi, A. Bouaricha, and A. Carle, *Computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation*, Tech. Report 1, Argonne National Laboratory, 1996.

[BCHK98] C. Bischof, A. Carle, P. Hovland, P. Khademi, and A. Mauer, *ADIFOR 2.0 user's guide (revision D)*, Argonne National Laboratory, March 1995. Revised: June, 1998.

[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.

[Bra94] A. Brandstädt, *Graphen und Algorithmen*, Leitfäden und Monographien der Informatik, B. G. Teubner, Stuttgart, 1994.

[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.

[BuFa89] R. L. Burden and J. D. Faires, *Numerical analysis, fourth edition.* PWS-KENT Publishing Company, Boston, Mass., 1989.

[CDB96] B. Christianson, L. Dixon, and S. Brown, *Sharing storage using dirty vectors* in [BBCG96], pp. 107–115.

[Chri93] B. Christianson, *Reverse accumulation of functions containing gradients*, Tech. Report 278, NOC, University of Hertfordshire, Hatfield, UK, 1993.

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

[CGRW92] G. Corliss, A. Griewank, T. Robey, and S. Wright, *Automatic differentiation applied to unsaturated flow — ADOL-C case study*, Preprint ANL/MCS–TM–162, Mathematical and Computer Science Division, Argonne National Laboratory, 1992.

[CoMo84] G. Corliss and J. More, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM Journal on Numerical Analysis, 20 (1984), pp. 187–209.

[CPR74] A. Curtis, M. Powell, and J. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.

[DaPr90] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*, Cambridge University Press, 1990.

[DPS90] P. H. Davis, J. D. Pryce, and B. R. Stephens, *Recent developments in automatic differentiation*, in J. C. Mason, and M. G. Cox (eds.), Scientific Software Systems, Chapman and Hall, 1990, pp. 166–182.

[FaPa97] C. Faure and Y. Papegay, *O∂yssée version 1.6 - the user's reference manual*, INRIA research report, INRIA, France, 1997.

[GaJo79] M. R. Garey and D. S. Johnson, *Computers and intractability - a guide to the theory of NP-completeness*, W. H. Freeman and Company, San Francisco, 1979.

[GUG96] U. Geitner, J. Utke, and A. Griewank, *Automatic computation of sparse Jacobians by applying the method of Newsam and Ramsdell* in [BBCG96], pp. 161-172.

[GUW97] A. Griewank, J. Utke, and A. Walter, *Evaluating higher derivative tensors through univariate Taylor polynomials*, Preprint, IOKOMO-09-1997, TUD, 1997.

[Gri91] A. Griewank, *The chain rule revisited in scientific computing*, SIAM News, 24 (1991), no. 3, p. 20 & no. 4, p. 8.

[GJU96] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, a package for automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, 22 (1996), pp. 131-167.

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

[Har69] F. Harary, *Graph Theory*, Addison-Wesley, 1969.

[HoSa78] E. HOROWITZ AND S. SAHNI, *Fundamentals of computer algorithms*, Computer Science Press, Inc., 1978.

[Iri91] M. IRI, *History of automatic differentiation and rounding error estimation* in [CoGr91], pp. 3-16.

[Kar75] R. M. KARP, *On the complexity of combinatorial problems*, Networks 5, 45-68, 1975.

[Knu92] D. E. KNUTH, *Literate programming*, CSLI Lecture Notes, no. 27, xiv+368 pp, Center for the Study of Language and Information, Stanford, University of Chicago Press, 1992.

[Kub91] K. KUBOTA AND M. IRI, *Estimates of rounding errors with fast automatic differentiation and interval analysis*, Journal of Information Processing, 14 (1991), pp. 508-515.

[Kub96] K. KUBOTA, *PADRE2-Fortran precompiler for automatic differentiation and estimates of rounding errors*, in [BBCG96], pp. 367-374.

[LeKn90] S. LEVY AND D. E. KNUTH, *The* `CWEB` *system of structured documentation*, Report STAN-CS-90-1336, 200 pp., Department of Computer Science, Stanford, 1990.

[Lin65] S. LIN, Bell System Technical Journal, Vol. 44, pp. 2245-2269.

[Lyn75] J. A. LYNCH, *The equivalence of theorem proving and the interconnection problem*, ACM SIGDA Newsletter 5:3, 1975.

[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.

[Nau94] U. NAUMANN, *Globale Optimierung mit Ergebnisverifikation auf Multiprozessorsystemen*, Diplomarbeit, TU Dresden, 1994.

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

[PTVF92] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, AND B. P. FLANNERY, *Numerical recipes in C*, Cambridge University Press, 1992.

[RaCo96] L. RALL AND G. CORLISS, *An introduction to automatic differentiation*, in [BBCG96], pp. 1-18.

[RoTa78] D. J. ROSE AND R. E. TARJAN, *Algorithmic aspects of vertex elimination on directed graphs.* SIAM Journal of Applied Mathematics, Vol. 34, no. 1, January 1978, 176-197.

[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.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda, *Methods for visual understanding of hierarchical systems*, IEEE Trans. Syst. Man Cybern., SMC-11(2):109-125, 1981.

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

Erklärung

gemäß §5(1) Pkt. 5a und 5b der Promotionsordnung

5a Versicherung: Hiermit versichere ich, daß ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind also solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Affirmation: I affirm that I have written this dissertation without any inadmissible help from any third person and without recourse to any other aids; all sources are clearly referenced. The dissertation has never been submitted in this or similar form before, either in Germany or in any foreign country.

5b Die vorgelegte Dissertation wurde am Institut für Wissenschaftliches Rechnen der Technischen Universität Dresden unter der wissenschaftlichen Betreuung von Herrn Prof. Andreas Griewank angefertigt.

This dissertation was written at the Institute of Scientific Computing, Dresden University of Technology under the scientific supervision of Prof. Andreas Griewank.

Dresden, den 26. November 1998