

RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LuFG Informatik 12 (Prof. Dr. rer. nat. Uwe Naumann)



**Exploiting Compiler Optimization on Efficient
Adjoint Code Generated from
Linearized Directed Acyclic Graphs
Master's Thesis**

Angelika Schwarz (287360)

December 4, 2014

Declaration of Compliance

I, the undersigned, declare that my term paper on the topic of “Exploiting Compiler Optimization on Efficient Adjoint Code Generated from Linearized Directed Acyclic Graphs” was penned autonomously, and that I have not used any other sources or means other than those mentioned in my paper. When using other sources – including electronic media – in wording or meaning, I have indicated this with a clear reference to the source. I acknowledge that in the case of an infringement, my term paper will be assessed as “failed” and that I will not receive a certificate. In addition, I am aware that non-compliance can result in further consequences. I agree that my paper may be reviewed by means of a search engine for plagiarism. I am aware of the fact that such a search engine can store my paper on a server. With my signature, I also assert that I have read the leaflet “Ethical Guidelines for the Authoring of Academic Work” adopted by the Department of Computer Science at the RWTH Aachen University from April 2007.

Abstract

This work presents an offline optimization procedure for static C++ code employing the vertex elimination Algorithmic Differentiation (AD) approach to generate first-order adjoint source code. Different heuristics have been implemented to determine an elimination sequence such that the number of floating-point multiplications is minimized. In addition to the traditional Markowitz-based heuristics, average path count and structural path count are two new heuristics that attempt to identify and eliminate graph substructures optimally.

The code generator produces adjoint codes whose structure is well-suited to make the compiler maximize the performance of the derivative code. Key aspect of compiler optimization is the improvement of memory accesses, which accounts for the memory hierarchy of the target platform. On the test platforms, the compiler freely reorders instructions and thereby improves the cache and register utilization. Apart from very specific examples, the impact of the chosen elimination strategy takes a back seat to the efficacy of compiler optimization.

The two-step optimization procedure of first vertex elimination and then compiler optimization promises to generate highly efficient adjoint code. This opens up the opportunity to outsource expensive, though static code segments of a program which is other than that differentiated via AD by operator overloading. A test case demonstrates the feasibility of this approach using the AD tool `dco/c++`.

Contents

1	Introduction	9
1.1	Motivation and Applications	9
1.2	Task and its Context	10
1.3	Related Work	11
1.4	Thesis Organization	12
2	Background	13
2.1	Techniques of Differentiation	13
2.2	Dimensions of Automatic Differentiation	15
2.2.1	Modes of Automatic Differentiation	15
2.2.2	Operator Overloading versus Source Code Transformation	16
2.3	Foundations of Vertex Elimination	17
3	Contribution	23
3.1	DAGs from Static Code	24
3.2	Elimination Strategies	27
3.2.1	Natural and Reverse	27
3.2.2	$1 : n$	28
3.2.3	Markowitz	30
3.2.4	Path Count	33
3.3	Code Generation	39
3.4	Helmholtz Example	43
3.5	Case Study: Effectiveness of Compiler Optimization	45
3.6	Integration into dco/c++	50
4	Conclusion	55
	Bibliography	57
A	Case Studies	61
A.1	Speelpenning's Example	62
A.2	Reverse Osmosis	65
A.3	Performance and Optimization Flags	68
B	User Documentation	71
C	Developer Documentation	77

1

Introduction

The evaluation of derivatives of some objective function is frequently a crucial component in scientific applications, especially numerical simulations. Typically, such numerical simulations involve optimization, partial differential equations, or the solution of systems of nonlinear equations – what all these tasks have in common is the need for derivatives. Section 1.1 gives some sample applications. The evaluation of derivatives, however, may turn out to be the dominant factor with respect to the total runtime [21]. Section 1.2 outlines the idea that is pursued in this thesis to reduce the time spent on the derivative computation. Section 1.3 presents related work and its differences to what is done here.

1.1 Motivation and Applications

Applications from meteorology [11], geophysics [6] or design optimization [25] yield large-scale simulations. These applications may at some point evoke one of the following problems.

- **Systems of Nonlinear Equations.** A classical task in numerical analysis is to find the roots of a vector-valued function $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$. One well-known algorithm solving this task is Newton’s method. It is a derivative-based method that successively approximates points where $F(\mathbf{x}) = 0$ [21].
- **Sensitivity Analysis.** Computer models can be analyzed with respect to the sensitivity to certain parameters, i.e. the impact of small perturbations of a certain input on the overall output. A small perturbation of an input parameter is propagated through the system. Its effect is then evaluated by the deviation of the original output and the perturbed output. Thereby, it can, for example, be estimated how susceptible to errors a model is. This may

especially be useful for measuring the sensitivity of the application to poorly known parameters (cf. [4, 11]).

- **Numerical Optimization.** Many applications require the minimization of an objective function $F: \mathbb{R}^n \rightarrow \mathbb{R}$. The goal then is to solve $\min_{\mathbf{x} \in \mathbb{R}^n} F(\mathbf{x})$. Prominent methods for this task harness derivatives to compute descent directions. This includes the method of steepest descent, which requires only first-order derivatives, and Newton’s method, which, in contrast, requires both first-order and second-order derivatives [21].

The objective functions that implement any of the applications from above are typically present in the form of a computer program. Although a couple of applications from the previous section require higher-order derivatives, the focus from now on is on first-order derivatives. Therefore, the term derivatives implicitly means first-order derivatives.

1.2 Task and its Context

This section is intended to give only a rough idea and is therefore intentionally kept short. The next chapter delves into background information on Algorithmic Differentiation (AD) and vertex elimination.

From a user’s perspective, derivatives of a function F with $\mathbf{y} = F(\mathbf{x})$ are to be computed in reverse mode AD. The AD tool to go with is `dco/c++`, see [16] for further details. The reverse mode of AD computes adjoint projections $\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$, which demands the reversal of the data flow. For this purpose, `dco/c++` uses a tape to record data that is computed in the forward run and that is required in the reverse run. In the reverse run, the tape is then interpreted.

Some section of the code, typically the body of a computationally expensive loop, should be treated separately. Such a code block, here denoted by f with $\hat{\mathbf{y}} = f(\hat{\mathbf{x}})$, is supposed to be treated as a black box procedure within `dco/c++`. Rather than being differentiated f with `dco/c++`, f is outsourced. In the forward run, f is not recorded in the tape; in the reverse run, an external implementation of the adjoint code of f is called. The local adjoints $\hat{\mathbf{x}}_{(1)}$ of f must be incorporated into the tape so that the global adjoints $\mathbf{x}_{(1)}$ of F are computed correctly. Figure 1.1 visualizes the logical flow. Since f is not recorded, outsourcing f probably reduces the memory requirement. Furthermore, there is the potential for speedup if optimized adjoint code of f is used in the reverse run. Regarding convention, throughout this thesis F denotes the global function whereas f denotes the code block supposed to be outsourced.

In this thesis, optimized adjoint code for f is generated through an offline procedure. The code generation comprises two optimization steps. First, optimized source code is generated through vertex elimination on linearized directed acyclic graphs. Second, the compiler excels in optimizing the generated source codes. The reason for this is that these codes are static. The optimized object files are then called by `dco/c++` in the reverse run.

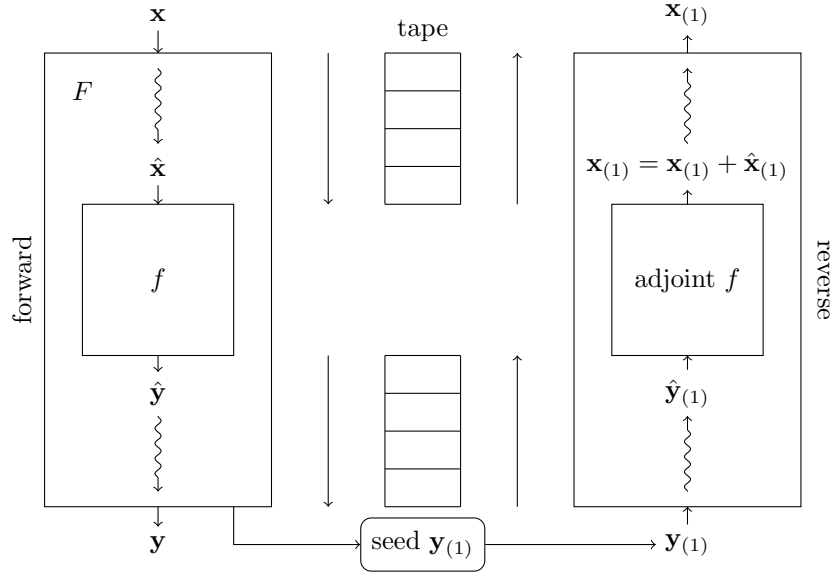


Figure 1.1: Reverse mode with `dco/c++` and f as an external function.

1.3 Related Work

Tadjouddine and others [10, 27, 28] have implemented a similar tool to generate first-order derivative code from linearized DAGs. Their development **EliAD**, **E**limination approach for **A**utomatic **D**ifferentiation, is a standalone AD tool for Fortran code. They, too, employ vertex elimination as technique to implement source code transformation.

Their starting point is the extended Jacobian system. The Jacobian matrix can be obtained by solving this sparse linear system by means of Gaussian elimination. In the process, fill-in¹ can be created, which can degrade the performance of the solver. Hence, the linear system ought to be solved while, at the same time, the fill-in is minimized. The latter can be accomplished by choosing an appropriate sequence of pivot elements. Although Tadjouddine et al. [10, 28] also utilize the graph interpretation, i.e. linearized DAGs with vertex elimination, rather than the matrix interpretation, i.e. the extended Jacobian matrix; they focus on Markowitz-related heuristics. Those heuristics were originally designed for minimizing the fill-in. However, the matrix interpretation with choosing a pivot element is equivalent to eliminating an intermediate vertex in the linearized DAG. The reason why Markowitz-related heuristics are nevertheless suited is that these heuristics also, as a side benefit, minimize the number of floating-point operations.

As described in [10], EliAD is written in Java and uses a front-end parser and a back-end pretty printer generated by ANTLR. The user specifies the independent variables and the dependent variables, respectively. This allows EliAD to perform a bidirectional data flow analysis. As a result, variables can be classified to be either active or passive. Once the input code has been read and the abstract syntax

¹zero entries in the original matrix filled with non-zero values

tree has been built up, EliAD differentiates each statement of the input program symbolically. Thereby, they generate large blocks of static code free of branches and loops.

In addition to minimizing the floating-point operation count through suited elimination sequences, Tadjouddine et al. [10] incorporate further code optimizations. Such optimizations include for instance the removal of trivial operations, e.g. multiplications with $+1$ or -1 . Furthermore, they identify cache and register utilization as a major factor in the improvement of performance. For this purpose, they manually reorder instructions. In [10] and especially [27] they analyze the effects of statement reordering on instruction-level parallelism and the time spent on memory accesses.

In contrast to EliAD, in this thesis several optimizations are intentionally left to the compiler. Compiler optimization handles propagation of constants and thereby also removes trivial operations such as multiplications with ± 1 . Furthermore, passive variables in the form of dead branches are removed. For this reason, the classification of variables as active or passive is deliberately omitted. Here, the starting point for the generation of optimized code is solely the reduction of floating-point multiplications. The optimization of memory accesses is not directly targeted, though indirectly addressed by compiler optimization. It turns out that the compiler indeed reorders statements – an aspect Tadjouddine et al. have recognized as essential for the performance and manually optimize.

1.4 Thesis Organization

The rest of this work is organized as follows. Chapter 2 establishes notation and introduces the traditional forward and reverse mode of AD and its connection to the vertex elimination approach of Griewank and Reese [12]. Based on this theoretic concept, Chapter 3 presents how the vertex elimination approach is used to generate adjoint source files as part of an offline optimization procedure. The major steps are as follows. Section 3.2 explains the vertex elimination strategies implemented here, according to which the adjoint source codes are generated. The example in Section 3.4 investigates the impact of the chosen elimination strategies on the efficiency of calculating the Jacobian matrix. Section 3.5 discusses the results of this comparison in light of compiler optimization. In Section 3.6 another test case reviews the potential for savings when the generated adjoint source files are employed within the AD tool `dco/c++` in order to outsource certain code segments. Finally, Chapter 4 presents conclusions.

2

Background

This chapter outlines several ways to evaluate derivatives, one of which is Algorithmic Differentiation (AD). To motivate the latter approach, Section 2.1 discusses advantages of AD over other techniques. Section 2.2 presents some aspects of AD in greater detail. In particular, forward and reverse mode are adumbrated and methods to implement these are introduced. Eventually, Section 2.3 introduces vertex elimination as an alternative approach to compute derivatives. Furthermore, there is a discussion of the relation of, on the one hand, vertex elimination and, on the other hand, forward and reverse mode of AD.

2.1 Techniques of Differentiation

To abstract from any particular example, throughout this thesis $F: \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\mathbf{y} = F(\mathbf{x})$, denotes some vector-valued function. The entries of $\mathbf{x}$ constitute the vector of *independent variables*, variables with respect to which a function is differentiated. The vector $\mathbf{y}$ denotes the *dependent variables*, output variables whose derivatives shall be computed. Since the inputs and outputs of F are vectors, the derivative is a matrix known as Jacobian matrix. For $F(\mathbf{x}) = (y_1(\mathbf{x}), y_2(\mathbf{x}), \dots, y_m(\mathbf{x}))^T$, the Jacobian matrix

$$\nabla F = \nabla F(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial y_1}{\partial x_n}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial y_m}{\partial x_n}(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m \times n}$$

aggregates the partial derivatives $\frac{\partial y_i}{\partial x_j}$ at point $\mathbf{x}$. For $m = 1$, there is only one single output that has to be differentiated with respect to all inputs. The Jacobian matrix then consists of only one row, which, written as a column vector, is referred to as the gradient of F .

Here are some approaches to accomplish the task of computing derivatives [4, 21].

- **Numerical Differentiation.** The value of a derivative can be approximated numerically. One method to approximate a column of the Jacobian matrix is finite differences. Consider for example first-order forward finite differences. The i -th column of the Jacobian matrix can be approximated using

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) = \frac{F(\mathbf{x} + h \cdot \mathbf{e}_i) - F(\mathbf{x})}{h}$$

where $\mathbf{e}_i$ is the i -th Cartesian basis vector. The accuracy of the approximation depends on the step size h . Choosing a suitable step size is a delicate matter because, on the one hand, the step size must be small to have a small approximation error and, on the other hand, must be large enough to avoid cancellation.

- **Symbolic Differentiation.** Computer algebra systems determine an explicit expression of the derivative of a function. [4] claims that symbolic differentiation is inefficient in terms of computing time since the number of underlying expressions can grow quickly.
- **Algorithmic Differentiation.** Every function given in the form of a computer program can be viewed as a composition of elementary mathematical functions. The number of elementary functions available in a programming language is limited. Since derivatives of these elementary functions are known, each statement of the code can be differentiated on its own. The results can be combined through the application of the chain rule. AD then defines techniques to evaluate the given program and its derivatives simultaneously up to machine accuracy in an automatic fashion. The associativity of the chain rule allows propagating derivatives in several ways, two of which are forward and reverse mode of AD.
- **Hand-Coded Derivatives.** Since the programmer knows about the structure of the program, derivatives implemented by hand can yield highly optimized code. The programmer knows about the control flow and therefore can interlace forward and reverse mode of AD wherever conducive. As the coding time grows with the program size, solely hand-written derivatives seem impracticable for simulation code. Furthermore, an obvious drawback is that it requires considerable human effort to maintain hand-written derivative code.

As conclusion to this section, the advantages of differentiating a program with AD include:

- **Flexibility.** AD can differentiate virtually any code. The approach works on both codes with only a few instructions and large codes comprising thousands of instructions. Furthermore, AD can deal with control structures such as branches or loops.
- **Accuracy.** The derivatives are exact to machine precision.

- **Computational cost.** Time and memory spent on the computation of derivatives should be minimized. Compared to the other approaches, AD excels in this regard.

The last point is addressed in this thesis. The goal is to scrutinize an alternative way of accumulating derivatives with regard to its potential for savings.

2.2 Dimensions of Automatic Differentiation

There are two basic approaches in AD, the so-called forward and the reverse mode. The difference between these modes is the way in which the chain rule is applied to propagate derivatives. Section 2.2.1 outlines how both approaches work. Forward and reverse mode can be implemented either by operator overloading or by source code transformation, which is briefly described in Section 2.2.2.

2.2.1 Modes of Automatic Differentiation

Consider a program to be given as

$$\begin{aligned} F(\mathbf{x}) &= F_{n-1}(F_{n-2}(\dots(F_1(F_0(\mathbf{x})))\dots)) \\ &= F_{n-1} \circ F_{n-2} \circ \dots \circ F_1 \circ F_0(\mathbf{x}) \end{aligned}$$

with each F_i being differentiable. The application of the chain rule yields

$$\nabla F = \nabla F_{n-1} \cdot \nabla F_{n-2} \cdot \dots \cdot \nabla F_1 \cdot \nabla F_0. \quad (2.1)$$

Due to the associativity of the chain rule, there are several ways to accumulate derivatives. Traditionally, AD implements two modes [21]. First, the *forward mode of AD* applies the chain rule in forward direction as illustrated in Figure 2.1. It calculates the value for each intermediate variable and its derivative with respect to the independent variables, i.e. the intermediate tangent-linear variables $\mathbf{v}_i^{(1)}$ hold the

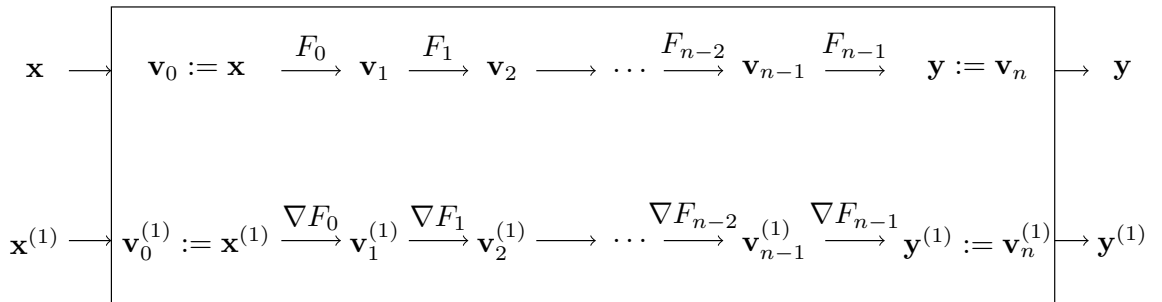


Figure 2.1: Tangent-linear model $\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ and the interpretation of tangent-linear variables in forward mode as $\mathbf{y}^{(1)} = (\nabla F_{n-1} \cdot \dots \cdot \nabla F_0) \cdot \mathbf{x}^{(1)}$.

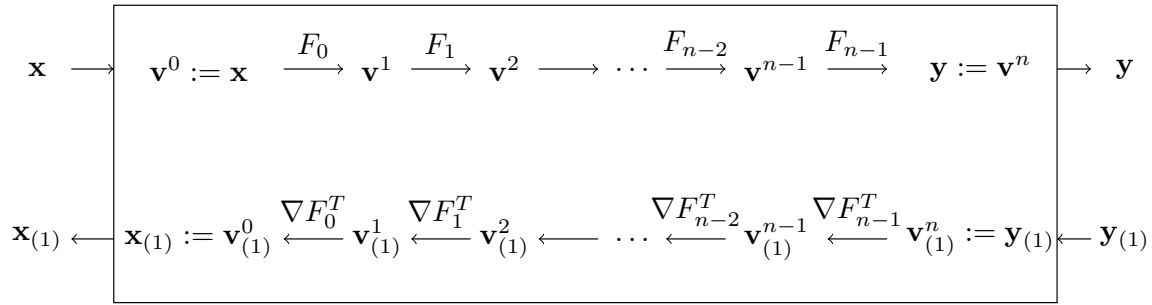


Figure 2.2: Adjoint model $\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$ and the interpretation of adjoint variables in reverse mode $\mathbf{x}_{(1)} = (\nabla F_{n-1} \cdot \dots \cdot \nabla F_0)^T \cdot \mathbf{y}_{(1)}$.

derivative of $\mathbf{v}_i$ with respect to $\mathbf{x}$. The computational cost of applying the chain rule in this way is proportional to the number of independent variables. The forward mode preserves the control flow of the program and, as a consequence, the original statements and the derivative statements can be interleaved.

The *reverse mode of AD*, by contrast, applies the chain rule in reverse direction and propagates so-called adjoints, derivatives of the dependent variables with respect to an intermediate variable. As the propagation commences with the dependent variables, the control flow of the program is reversed, which is shown in Figure 2.2. Intermediate adjoint variables $\mathbf{v}_{i(1)}$ hold the derivative of $\mathbf{y}$ with respect to $\mathbf{v}_i$. Values of intermediate variables that are required for the reverse sweep must be stored, for example in a tape, or recomputed. The computational cost of this model is proportional to the number of dependent variables.

2.2.2 Operator Overloading versus Source Code Transformation

Given a piece of code, a transformation computing the derivatives of that code is aimed for. There are two basic methods to implement this transformation in either forward or reverse mode [3, 21].

Operator Overloading

First, there is operator overloading. Here, a custom data type is constructed. Arithmetic operations ($+$, $*$, ...) and mathematical functions (`exp`, `sqrt`, ...) are redefined in a way such that they operate on this custom data type and compute the value of the operation alongside with the derivative [21].

Example 2.2.1. Let y , x_1 and x_2 be of a data type for the forward mode. The expression $y = x_1 \cdot x_2$ with an overloaded multiplication operator then does not only compute the product of x_1 and x_2 , but also the corresponding directional derivative $y^{(1)} = x_1^{(1)} \cdot x_2 + x_1 \cdot x_2^{(1)}$. The derivative variable $y^{(1)}$ associated with y is updated

according to the product rule. Each multiplication therefore affects an update of two variables.

Similar to the multiplication operator, all elementary operations can be augmented to work on the custom data type through the application of standard differentiation rules.

A requirement for this approach is that all relevant variables must be defined as variables of the custom data type. Though the usage is convenient, operator overloading also has a disadvantage. Despite advances in the compilation of operator overloading, compiler optimization is less effective in comparison with straight-forward, plain source code [4].

Source Code Transformation

The second method to implement AD is source code transformation. The source code is explicitly rewritten to code that contains the original expressions and corresponding automatic differentiation expressions in forward or reverse mode. Since program analysis or source transformation takes place at compile time, the program can be analyzed in its entirety. Therefore, AD preprocessors have access to the context and may use the associativity of the chain rule to produce better code [4, 13].

Source code transformation, however, also has some disadvantages [4]: Source code transformation tools require infrastructure (e.g. parsing or analysis infrastructure) just as traditional compilers. In particular, this is necessary to deal with language-specific features. Thus, the implementation of source code transformation tools for complex languages may turn out to be rather difficult.

2.3 Foundations of Vertex Elimination

This section introduces vertex elimination on linearized DAGs as an alternative method of computing derivatives. Eventually, vertex elimination is related to Equation (2.1) and thereby also to the forward and the reverse mode.

Once again, let $F: \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\mathbf{y} = F(\mathbf{x})$, be a function present as computer program. So far F has not been specified. From now on F is assumed to have neither loops nor branches. Furthermore, let F be given in *single assignment code* (SAC), which ensures that variables are never overwritten. Then F looks as follows.

$$\begin{aligned} v_i &= x_i & i &= 0, \dots, n-1 \\ v_j &= \varphi_j(v_i)_{i \prec j} & j &= n, \dots, n+p+m-1 \\ y_i &= v_{n+p+i} & i &= 0, \dots, m-1 \end{aligned}$$

An assignment $v_j = \varphi_j(v_i)_{i \prec j}$ means that the variable v_i occurs in the elemental function φ_j .

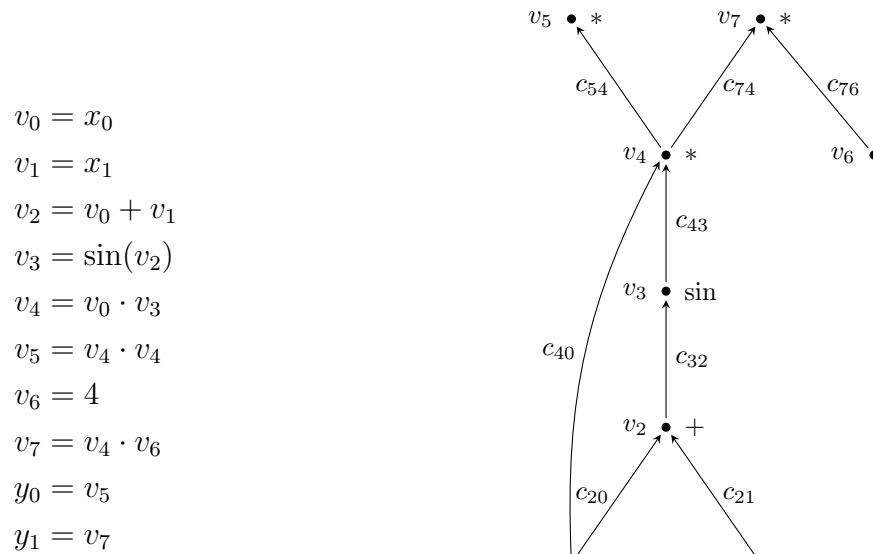
The *directed acyclic graph* (DAG) represents F as graph $G = (V, E)$. Each SAC variable v_i , $i = 0, \dots, n + m + p - 1$, turns into a vertex v_i . Edges indicate data dependencies between vertices, i.e. an edge from v_i to v_j , $i < j$, means that v_i contributes to the value of v_j . Labeling edges with local partial derivatives c_{ji} of the elemental functions *linearizes* the DAG. The partial derivative $\partial y_l / \partial x_k$ is computed as the sum of the products of all edge labels for all paths from x_k to y_l . Formal definitions of the SAC and the DAG can be found in [21].

Example 2.3.1. Consider the code fragment in Program 1. It implements a function that has two independent variables, x_0 and x_1 , and two dependent variables, y_0 and y_1 . The final goal is to compute the Jacobian matrix.

$$\begin{aligned} t &= \sin(x_0 + x_1) \cdot x_0 \\ y_0 &= t \cdot t \\ y_1 &= 4 \cdot t \end{aligned}$$

Program 1: Favorite example

Section 3.3 takes up this code fragment to exemplify how the code generation from the DAG actually works. Furthermore, Section 3.5 employs this “favorite” example to discuss effects coming from compiler optimization.



Program 2: SAC.

Figure 2.3: Linearized DAG.

Program 2 shows the SAC for this example, Figure 2.3 the corresponding linearized DAG. An entry $\partial y_i / \partial x_j$ of the Jacobian matrix can be accumulated by the product

of the edge labels on all paths that connect x_j and y_i . For instance, since there are two paths from $v_0 \equiv x_0$ to $v_7 \equiv y_1$, the partial derivative $\partial y_1 / \partial x_0$ is given by

$$\begin{aligned} \frac{\partial y_1}{\partial x_0} &= c_{20} \cdot c_{32} \cdot c_{43} \cdot c_{74} + c_{40} \cdot c_{74} \\ &= 1 \cdot \cos(v_2) \cdot v_0 \cdot v_6 + v_3 \cdot v_6. \end{aligned}$$

Having in mind that the entire Jacobian matrix should be accumulated, some edge labels are used multiple times. Hence, it makes sense to store the value of common subexpressions such as $c_{43} \cdot c_{32}$ to avoid multiple evaluations of the very same expression. If the result of $c_{43} \cdot c_{32}$ were stored in a temporary variable, the number of floating-point multiplications would reduce by three because this subpath contributes to each entry of the Jacobian.

Following the logic of accumulating local partial derivatives by multiplying along paths and adding the products, vertices can be *eliminated* whilst preserving the contribution of incident edge labels to the Jacobian matrix. If a vertex v_j is eliminated, for each pair of an incoming edge c_{ji} , $i \prec j$, and an outgoing edge c_{kj} , $j \prec k$, if required, a new edge c_{ki} is created and labeled with $c_{kj} \cdot c_{ji}$. If an edge c_{ki} already existed, this edge absorbs the new contribution $c_{kj} \cdot c_{ji}$ by adding it. Afterwards v_k and its incident edges can be removed. Figure 2.4 illustrates how the elimination of a vertex changes the linearized DAG.

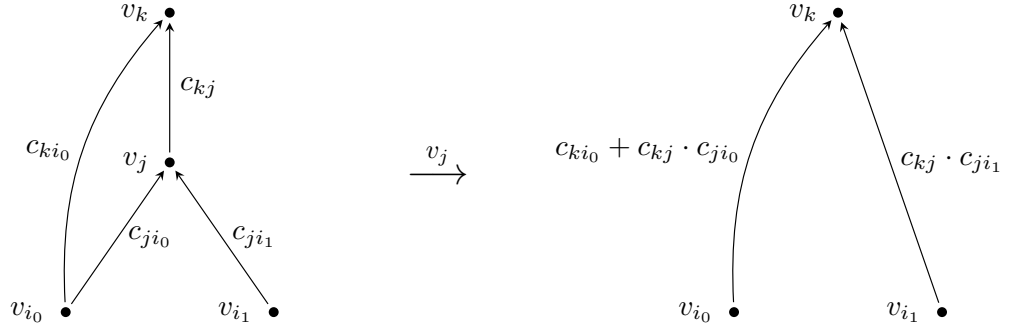


Figure 2.4: The elimination of v_j in the left DAG yields one absorption (left edge) and the creation of one new edge (right edge).

The order in which vertices are eliminated is called the *elimination sequence*. Ideally this elimination sequence exploits common subexpressions to reduce the overall operation count. If all vertices on paths between the independents and the dependents have been removed, the edges directly connecting independent vertices and dependent vertices are labeled with the partial derivatives. In literature, this state is reached when all intermediate vertices have been removed and the graph has been rendered bipartite. DAGs in this thesis are built up from real code or, in other words, code which is not necessarily present in single-assignment form. As a result, non-SAC conform statements can be encountered like for instance read accesses to dependent variables. Even if in this scenario only dependent and independent vertices remain after having processed the elimination sequence, the final DAG is not

bipartite. Thus, the notion of “eliminating intermediate vertices” does not seem appropriate. Constant vertices are not eliminable and neither are vertices that have a dual role as intermediate vertex and dependent vertex. The remainder of this thesis therefore uses the term “eliminable vertices” rather than “intermediate vertices”.

At the beginning of Section 2.2.1, Equation (2.1) was used to introduce forward and reverse mode as two accumulation schemes. Forward mode solves the matrix product from right to left, whereas reverse mode employs the opposite order. In fact, Equation (2.1) allows a plethora of accumulation schemes with forward and reverse mode being an extreme choice each [13]. The remainder of this section addresses the connection between forward and reverse mode on the one hand, and vertex elimination on the other hand.

The graph interpretation and the matrix interpretation as in Equation (2.1) are equivalent. Based on [22], a function $\mathbf{y} = F(\mathbf{x})$ can be written as extended SAC

$$\mathbf{y} = Q_m \cdot \Phi_q(\dots(\Phi_1(\mathbf{v}_0)))$$

where $\mathbf{v}_i = \Phi_i(\mathbf{v}_{i-1})$ and $\mathbf{v}_0 = (v_{1-n}, \dots, v_0, 0, \dots, 0)$. The vector

$$\mathbf{v}_i = (\underbrace{v_{1-n}, \dots, v_0}_{\mathbf{x}}, v_1, \dots, v_p, \underbrace{v_{p+1}, \dots, v_{p+m}}_{\mathbf{y}})$$

maintains memory locations for all SAC variables. For an assignment $v_j = \varphi_j(v_i)_{i \prec j}$ the transformation Φ_j updates the entry v_j and leaves all other entries untouched. Hence, the vector $\mathbf{v}_i = (v_{1-n}, \dots, v_0, v_1, \dots, v_i, 0, \dots, 0)$ contains the values of all SAC variables that have been processed so far, i.e. up to v_i , and zeroes for all pending assignments. At last, the final m entries in $\mathbf{v}_q$ contain the function values, which are extracted by $Q_m = (0, \dots, 0, I_m) \in \mathbb{R}^{m \times n+p+m}$.

If the extended SAC is slightly rewritten, the connection to Equation (2.1) becomes apparent.

$$F(\mathbf{x}) = Q_m \cdot \Phi_q \circ \Phi_{q-1} \circ \dots \circ \Phi_1 \cdot (P_n^T \mathbf{x})$$

The matrix $P_n^T = (I_n, 0, \dots, 0)^T \in \mathbb{R}^{n+p+m \times n}$ simply aligns the dimensions for the matrix product. Thus, the Jacobian matrix can be computed from the product

$$\nabla F = Q_m \cdot J_q \cdot J_{q-1} \cdot \dots \cdot J_1 \cdot P_n^T$$

of elemental Jacobians $J_j := \partial \Phi_j / \partial \mathbf{v}_{j-1}$. Such an elemental Jacobian aggregates the local partial derivatives known from the linearized DAG, i.e.

$$J_j = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ c_{j0} & \dots & c_{jj-1} & 0 & & & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \end{pmatrix}.$$

The product of elemental Jacobians comprises sparse lower triangular matrices, which can be bracketed and pruned. Brackets provide an opportunity to find better accumulation schemes than the obvious forward and reverse mode. Here, the number of floating-point multiplications is used to assess how good an accumulation scheme is. The task of finding a bracketing scheme that minimizes the number of multiplications can be solved by dynamic programming. Commutativity of certain matrices, however, adds another dimension. Matrices that do not lie on the same path in the commute with each other, which opens up even more accumulation schemes not incorporated in the dynamic programming solution. Vertex elimination addresses this problem. As a consequence, codes generated from vertex elimination typically require less floating-point operations to accumulate the Jacobian matrix than the traditional forward and reverse mode, respectively.

3

Contribution

By means of vertex elimination, derivative code shall be generated from linearized directed acyclic graphs (DAGs). In doing so, the number of floating-point multiplications serves as cost model to measure the quality of the generated code. Vertex elimination therefore aims at finding an elimination sequence that minimizes the cost of the generated derivative code. Unfortunately, if all eliminable vertices shall be removed, there are as many elimination sequences as there are permutations of eliminable vertices.

The first half of this chapter focuses on heuristics to determine suited elimination sequences and its concomitant code generation. DAGs in this chapter are built up by overloading; Section 3.1 briefly explains how the DAGs are created. Section 3.2 then introduces the heuristics employed here to find a good elimination sequence. Each subsection presents the underlying idea and differences, if any, between the version known from literature and the version actually implemented. For the most part, elimination strategies involve a couple of variants. Section 3.3 focuses on the code generation and describes how the generated source files look like. Eventually, the performance of source codes generated from different elimination strategies is compared in Section 3.4 and related to traits of compiler optimization in Section 3.5.

The second half of this chapter tackles the interaction of `dco/c++` and the generated source codes. Section 3.6 explains how a code segment can be outsourced and differentiated with the code generator, while the remainder of the program is differentiated with `dco/c++`. Finally, a test case demonstrates that this approach can result in speedup.

3.1 DAGs from Static Code

In Section 2.3 DAGs have been assembled from single assignment code (SAC). DAGs in this chapter look slightly different because they are built up from real code, which includes the potential of variable overwrites, read accesses to dependent variables, loops and branches. Furthermore, the code generation is part of an offline optimization procedure, which imposes restrictions on the source code to be differentiated. The prerequisite for loops and branches is that conditions and loop parameters must be known at code generation time. DAGs in this chapter are drawn upside down to distinguish them from the DAGs introduced in Chapter 2.

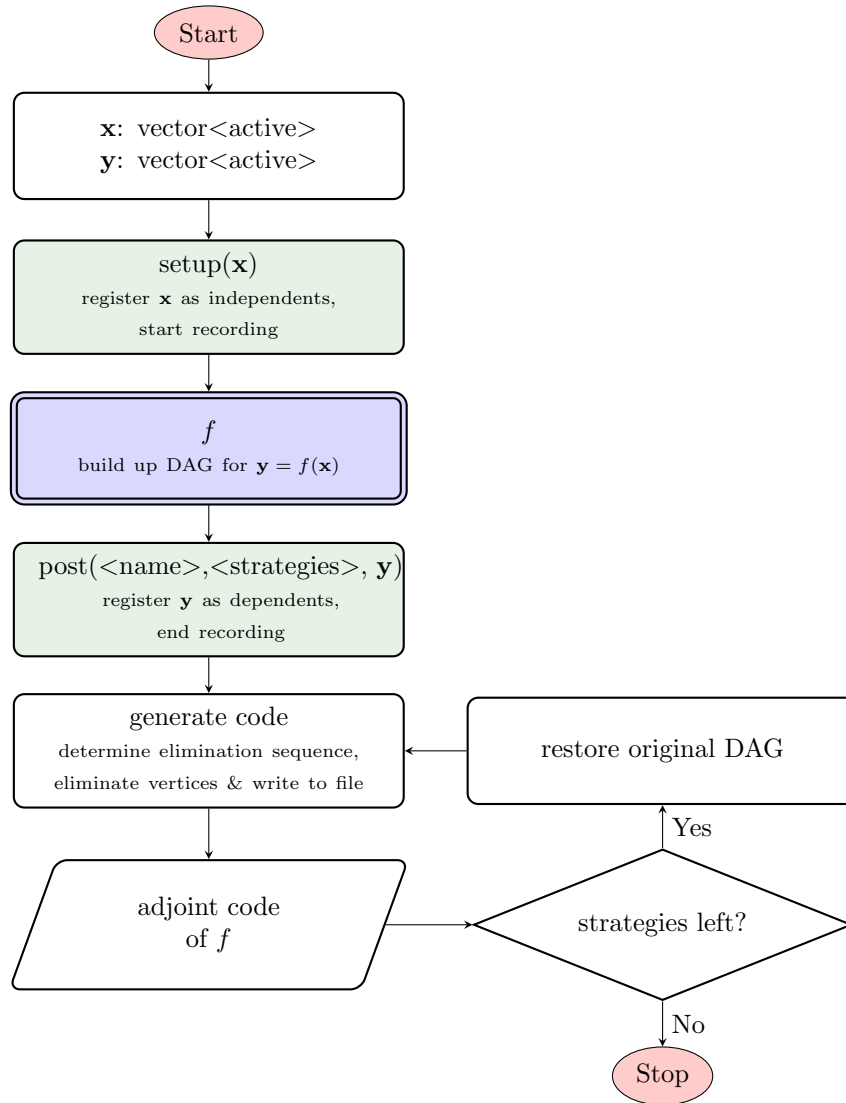


Figure 3.1: Program structure for code generation.

Operator overloading with an `active` data type is used to build up DAGs. Figure 3.1 shows how the generation of the adjoint codes works. Any code aside from the subprogram f is not required and therefore skipped in the illustration. The functions `setup(vector<active> &x)` and `post(string name, int strategies,`

`vector<active> &y)` enclose the subprogram for which derivative code shall be generated. Any operation between the two function calls adds vertices respectively edges to the DAG. The vector of independent variables `x` and the vector of dependent variables `y` must be explicitly registered by the user. The function parameter `strategies` specifies the elimination strategies to be employed in the code generation process. If multiple strategies are specified, the DAG is reset before the next strategy is applied. The function parameter `name` defines the base name assigned to the generated source files. The name is extended by appending a number to identify the used elimination strategy.

During code generation, the subprogram f does not compute any results. Consequently, the input vector `x` does not have to be initialized. The adjoint source files are entirely assembled from symbolic computations – so to speak, operator overloading is utilized to implement source code transformation.

In the DAG, vertices are numbered consecutively, $v_0, v_1, \dots$. Similarly, edges are labeled with ascending numbers $c_0, c_1, \dots$ rather than using the vertex IDs c_{ij} . As the independent variables may or may not correspond to the first n vertices $v_0, \dots, v_{n-1}$, the independent vertices are explicitly labeled as such with x_i , $i = 0, \dots, n-1$. Dependents, just as independents, are explicitly marked as y_i , $i = 0, \dots, m-1$.

When building up the DAG from real code, some of the following issues may be encountered.

- **Branches and loops.** Provided that the program always exhibits the same control flow, loops and branches can be used in the implementation. While the DAG is built up, branch conditions are evaluated, loops are unrolled. As a result, the control flow is fixed. Thus, in contrast to e.g. the parsing approach of EliAD, there is no explicit knowledge about the program structure and loops or branches cannot be recognized as such.
- **Independent vertices not being source vertices.** Any computation enclosed with `setup()` and `post()` creates vertices and edges. If there is some computation before the vector of independents is accessed, this computation nonetheless creates vertices and edges. Consider for example the initialization of independent variables. A vertex is created for the value. When the value is assigned to an independent variable, an ingoing edge is added to the independent vertex. For the adjoint source files, anything beyond scope yields dead code, which is ultimately removed by the compiler.
- **Read accesses to dependent variables.** Each read access to a dependent variable adds one outgoing edge to the corresponding vertex. Such vertices have a dual role. On the one hand, they represent a dependent variable and therefore must not be eliminated. On the other hand, they represent an intermediate result, which contributes to other dependent variables. When the elimination sequence is determined according to an elimination strategy, only the structure of the DAG, but not the actual role of certain vertices is taken into account. Thus, a dependent vertex can occur in an elimination sequence. There are two ways to deal with such a situation. First, an elimination sequence can be determined once at the beginning and is then processed step by

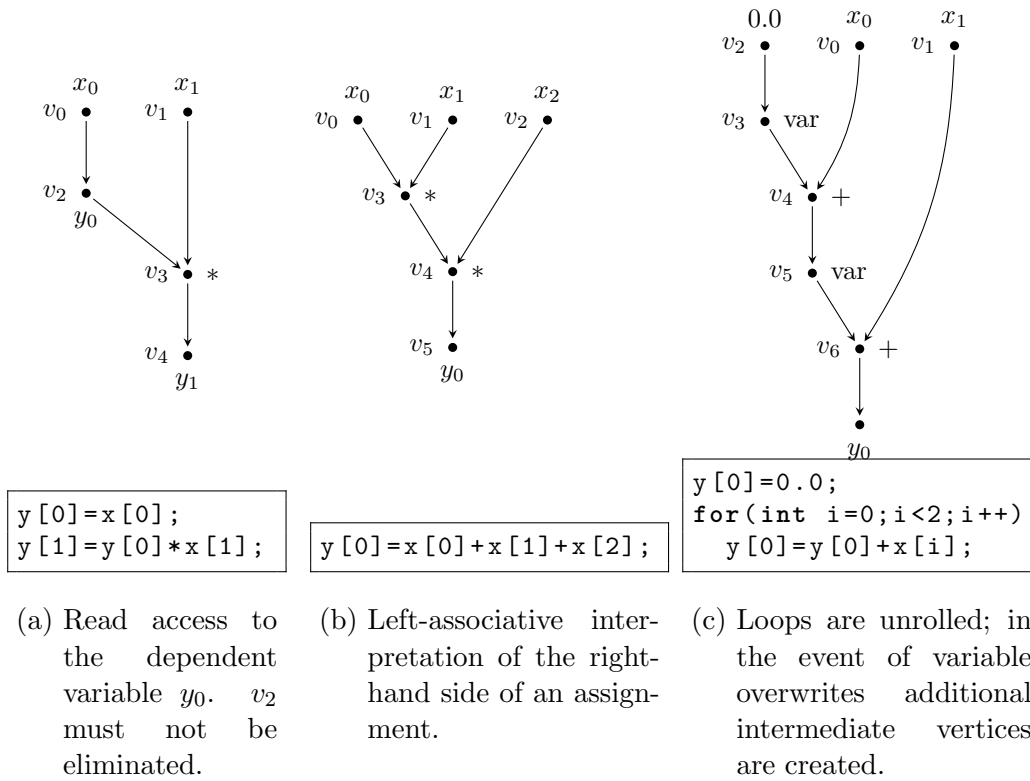


Figure 3.2: Code snippets not in SAC form and their DAGs.

step. Only when a vertex is about to be eliminated, it is checked if the vertex is per se excluded from elimination. If so, the vertex is skipped. Second, rather than computing an entire elimination sequence, an elimination strategy selects only one vertex at a time, namely the vertex with the currently best assessment. The assessment of vertices excluded from elimination is manipulated so that these vertices are never chosen for elimination. Depending on the type of elimination strategy, the problem with dependent variables is solved in either of the two ways. See Figure 3.2a for an example of a read access to a dependent variable.

- **Variable overwrites.** Variable overwrites are implicitly rewritten into SAC form because a new intermediate variable is created for each assignment operator. Figure 3.2c illustrates this with an example.
- **Assignments with multiple operations on the right-hand side.** Complex right-hand sides are decomposed into a sequence of operations. Intermediate vertices are created for each binary or unary operation. Figure 3.2b gives an example.

Example 3.1.1. Figure 3.3 shows the DAG for the favorite example, which is used later on to exemplify how the implemented elimination strategies and the code generation work.

Due to operator overloading, each assignment of a value to a dependent variable creates one additional intermediate vertex. The corresponding local partial deriva-

tives c_7 respectively c_{10} are equal to 1. Furthermore, the square operation adds two outgoing edges to v_4 respectively two incoming edges to v_5 .

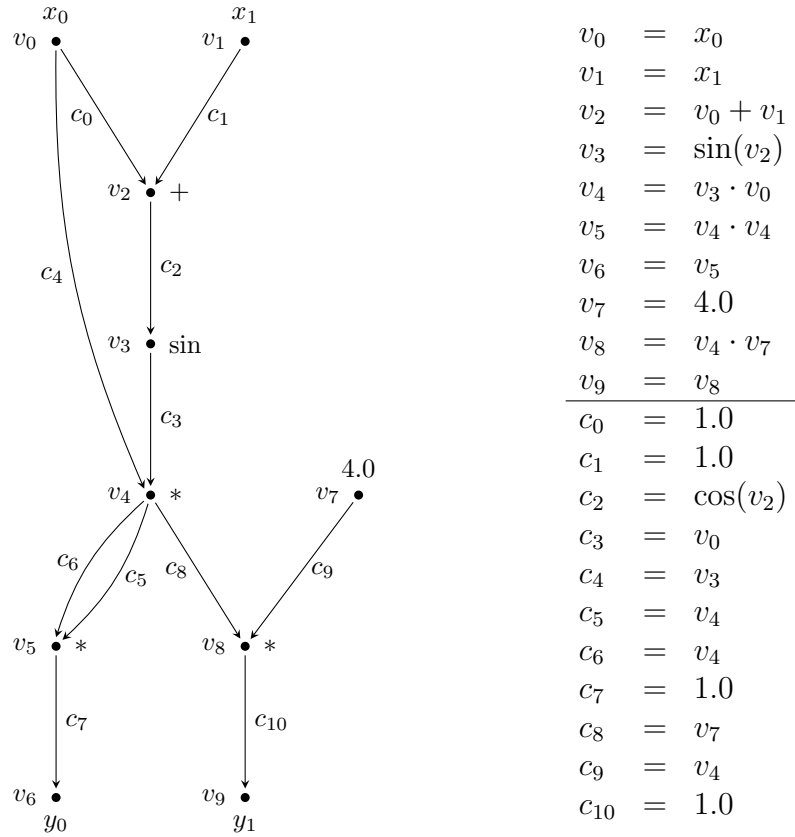


Figure 3.3: DAG for the favorite example as built up by overloading.

3.2 Elimination Strategies

This section describes the heuristics and their variants implemented and applied here. Apart from natural and reverse ordering, elimination strategies typically choose non-monotonic elimination sequences, which Griewank and Walther [13] refer to as cross-country elimination. In literature, the goal is usually the elimination of all eliminable vertices. Here, however, some elimination strategies intentionally do not eliminate certain vertices, namely those vertices whose elimination is costly.

3.2.1 Natural and Reverse

When the DAG is built up, each vertex is assigned an ID. The vertices are numbered consecutively in the order in which the vertices are created. If the vertices are eliminated in ascending order with respect to their IDs, the elimination strategy is referred to as *natural order*. If the opposite elimination order is chosen, the associated elimination strategy is called *reverse order*. The monotonic orderings

in either natural or reverse order are obviously closely linked to the conventional forward and reverse mode of AD. From a graph-based perspective, either ordering is likely to perform poorly on general DAGs because the structure of the DAG is not taken into account.

Example 3.2.1. Figure 3.4 and Figure 3.5 show how the elimination strategies natural order respectively reverse order are applied to the DAG of the favorite example.

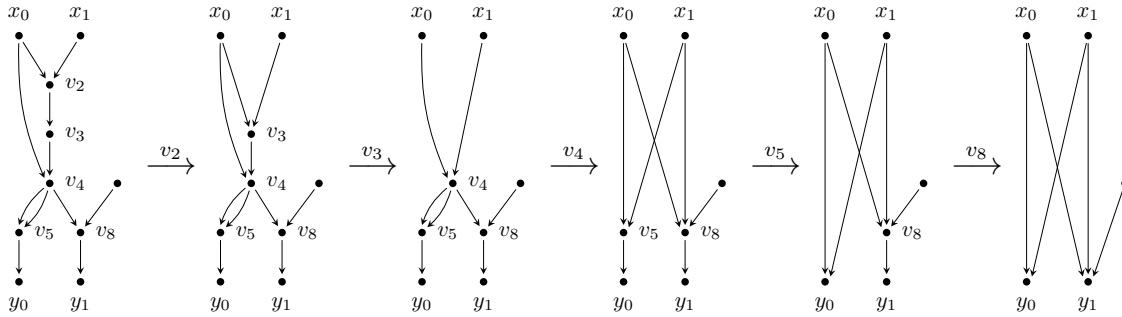


Figure 3.4: Elimination sequence following natural order.

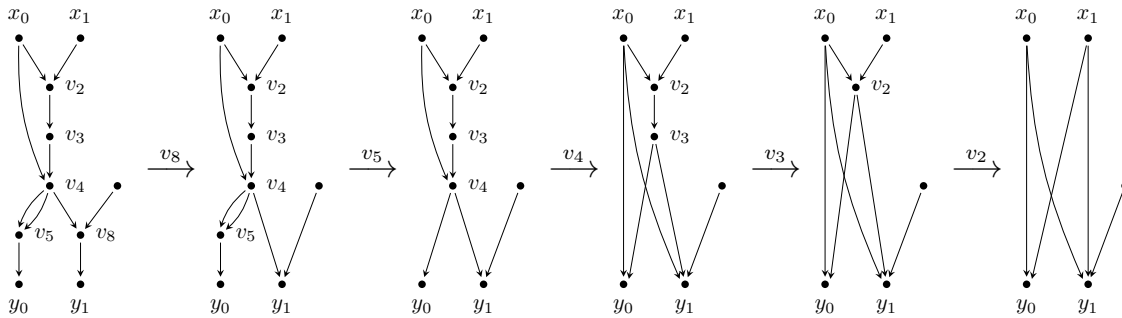


Figure 3.5: Elimination sequence following reverse order.

3.2.2 1 : n

For the elimination strategies covered below, vertices of DAGs are categorized according to the number of incoming and outgoing edges, respectively. Vertices that

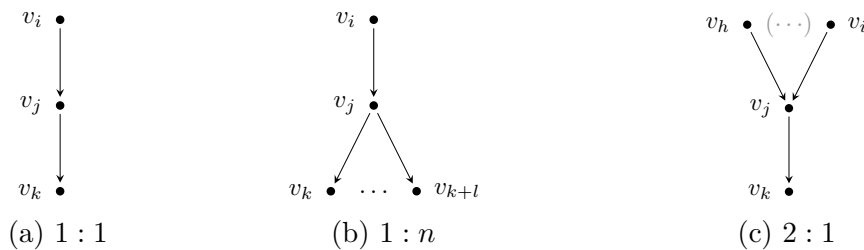
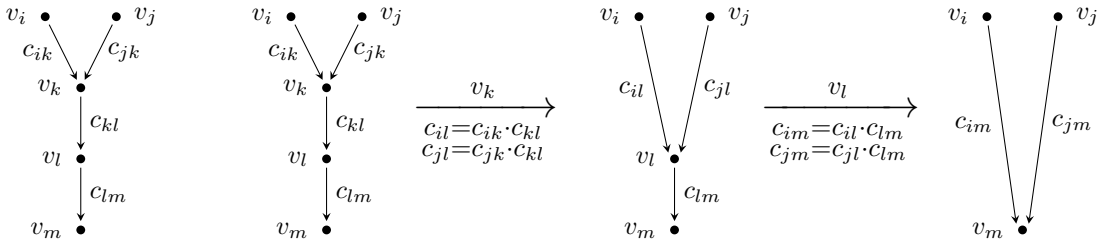


Figure 3.6: Vertex categories for safe eliminations from the perspective of v_j .

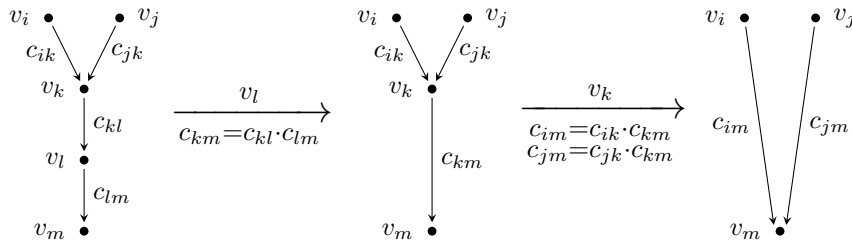
have exactly one incoming edge and exactly one outgoing edge are referred to as vertices of type 1 : 1. 1 : n denotes vertices with exactly one incoming edge and several outgoing edges. As in the original DAG before any vertices have been eliminated a vertex can have at most two predecessors, the third category comprises vertices of type 2 : 1 (rather than being called $n : 1$, though after the elimination of vertices a vertex may have more than 2 ingoing edges). For convenience reasons, remaining vertices are called $n : m$ vertices. Figure 3.6 illustrates the categories.

Due to unary and binary operations, DAGs tend to have a lot of 1 : 1 and 2 : 1 vertices. This observation motivates strategies that only take eliminable vertices of type 1 : 1, 1 : n , and 2 : 1 into consideration. In contrast to the elimination of $n : m$ vertices, any elimination pays for itself. Though the emerging DAG may still contain intermediate vertices, its size should be considerably smaller. Griewank and Walther [13] refer to these elimination as safe preeliminations.

Vertices of type 1 : 1 can obviously be eliminated without affecting the overall operation count. Figure 3.7 illustrates that the order in which 1 : 1 vertices and 1 : n respectively 2 : 1 vertices are considered has impact on the overall operation count. Optimality can only be preserved if 1 : 1 vertices are eliminated first, followed by the elimination of 2 : 1 and 1 : n vertices.



(a) Initial DAG, (b) Elimination of first v_j and then v_l creates four new edges and increases the (theoretical) multiplication count by four.



(c) The elimination sequence of first v_l and then v_k generates three new edges and yields a (theoretical) multiplication count of three.

Figure 3.7: Optimality can be preserved by eliminating 1 : 1 vertices first.

Now there are two options. First, the strategy can be used on its own. Safely eliminable vertices are easy to determine, which results in fast code generation. In the test cases, this strategy is referred to as 1 : n . Second, the strategy can be used as a preprocessing step for other strategies.

Regarding the strategy $1 : n$, vertices of type $2 : 1$ or $1 : n$ are considered to be equally worthwhile being eliminated. Thus, once all $1 : 1$ vertices have been eliminated, the exact order in which vertices of type $1 : n$ or $2 : 1$ are eliminated depends on a tiebreaker. Vertices of type $n : m$, by contrast, are never eliminated. The case studies confirm that this idea is indeed a viable strategy. The strategy $1 : n$ does not per se fall behind other strategies. This is a consequence of the way the DAG is built up. More precisely, most vertices fall into either of the categories $1 : 1$ or $2 : 1$. Example 3.3.1 demonstrates how the code generation works using $1 : n$ on the favorite example.

If an elimination criterion allows several vertices to be eliminated in the next step, a tiebreaker resolves the draw. Here, the vertex ID, or, in other words, the topological ordering of the DAG vertices fulfills this very role. The tiebreaker ‘natural’ selects the vertex with the smaller ID; analogously, ‘reverse’ gives preference to the vertex with the larger ID. In case of tiebreaker ‘random’, anything can happen.

A variation of the strategy $1 : n$ is termed *triangles*. The strategy *triangles* switches between tiebreakers depending on the category ($1 : 1$, $1 : n$, $2 : 1$) a vertex belongs to, whereas the strategy $1 : n$ sticks to one tiebreaker. As a first step, all vertices of type $1 : 1$ are eliminated. Next, all vertices of type $1 : n$ are eliminated in natural order. Last, all vertices of type $2 : 1$ are eliminated in reverse order. Consequently, no tiebreaker is required. The remaining vertices are again left untouched.

3.2.3 Markowitz

The Markowitz criterion, in its original form, targets the fill-in problem occurring in Gaussian elimination. The fill-in should be kept small for sparse matrices because any zero entry turning into a non-zero entry must be removed later in the elimination process, which may degrade the performance. The produced fill-in can be reduced by choosing a good pivot sequence. One of the first heuristics tackling this problem was presented by Markowitz in 1957 [17].

Already in Section 1.3 on related work in the form of EliAD, it was stated that, as a side benefit, the Markowitz criterion also minimizes the number of multiplications. As studied in [12], this makes the Markowitz criterion applicable to determining elimination sequences for the vertex elimination approach. The Markowitz degree of a vertex v_j

$$\mu(v_j) = |\{v_i : v_i \prec v_j\}| \cdot |\{v_k : v_j \prec v_k\}|$$

is an upper bound for the number of edges newly generated through the elimination of vertex v_j and thereby also an upper bound for the number of multiplications induced by the elimination of v_j . Local assessment of a vertex in form of its Markowitz degree can then be used to determine an elimination sequence.

Vertices with a small Markowitz degree promise little growth of the edge count through newly created edges, or equivalently, little increase in the multiplication count. Hence, vertices with small Markowitz degree should be eliminated first. The elimination strategy *static minimum Markowitz* computes the Markowitz degrees of

all eliminable vertices in the original graph and, based on this assessment, eliminates the vertices in ascending order with respect to their Markowitz degree. This yields an elimination sequence up to ties, i.e. vertices that have an equal Markowitz degree. A tiebreaker is required to resolve such draws.

As the immediate neighbors of a vertex v_j determine its Markowitz degree, the Markowitz degree is likely to change whenever a successor or a predecessor of v_j is eliminated. The strategy termed *dynamic minimum Markowitz* takes this into account. In order to keep the assessment of vertices up-to-date in the elimination process, the Markowitz degree is recomputed at each elimination stage. Once again, the vertex with the minimum Markowitz degree is chosen for elimination. A tiebreaker is needed if several vertices have an equal Markowitz degree. The dynamic variant of minimum Markowitz is potentially expensive because of the repeated computation of the Markowitz degrees. Since the search for an elimination sequence and the code generation are part of an offline procedure, time spent on recomputations is not a concern here.

An improvement to the Markowitz degree proposed by Naumann [20, 22] takes the interdependence of vertices into account. The elimination of a vertex influences the cost of the elimination of other vertices. As a result, the Markowitz degree of a vertex fluctuates in the course of elimination. Thus, there is the choice between (a) eliminating the vertex at this point in time and (b) postponing the elimination. Alleviating this task, (a) the cost of eliminating the vertex now (given by its current Markowitz degree) and (b) the cost of eliminating the vertex last in the elimination sequence is considered. The latter is expressed by the dependence degree of a vertex. *Vertex lowest relative* uses the difference between the current Markowitz degree and the dependence degree to determine an elimination sequence.

$$\hat{\mu}(v_j) = \mu(v_j) - \underbrace{\iota(v_j) \cdot \delta(v_j)}_{\text{dependence degree}}$$

Sorting the relative Markowitz degrees $\hat{\mu}(v)$ in ascending order gives the elimination sequence. The dependence degree is defined as the product of the number of independent vertices to which v_j is connected and the number of dependent vertices to which v_j is connected.

$$\begin{aligned} \iota(v_j) &= |\{v_k : v_k \text{ is independent vertex} \wedge v_k \prec^+ v_j\}| \\ \delta(v_j) &= |\{v_k : v_k \text{ is dependent vertex} \wedge v_j \prec^+ v_k\}| \end{aligned}$$

The variant of vertex lowest relative implemented here is termed *static relative Markowitz*. It computes the relative Markowitz degrees statically to determine an elimination sequence. For this purpose, once again a tiebreaker is required.

So far Markowitz-based strategies remove all eliminable vertices. Considering the increase in the multiplication count, it may be beneficial to intentionally not eliminate vertices with several predecessors and several successors. *Threshold Markowitz* takes up this idea, as illustrated in Algorithm 1. At first, safe eliminations in the form of applying $1 : n$ reduce the size of the DAG. Next comes the assessment of

Algorithm 1 Threshold-Based Minimum Markowitz**Input:**

- DAG with a set of vertices V
- set of independent vertices $V_{indep} \subseteq V$, set of dependent vertices $V_{dep} \subseteq V$
- set of elimination candidates determined as

$$V_e = V - (V_{dep} \cup V_{indep} \cup \{v \in V \mid v.role \equiv constant\})$$
- threshold t

```

1: apply strategy 1 : n;
2: let  $V_e$  be the updated set of remaining eliminable vertices
3: while  $\{v \in V_e \mid \mu(v) \leq t\} \neq \emptyset$  do
4:   eliminate  $v'$  with  $\mu(v') = \min\{\mu(v) \mid v \in V_e\}$ 
5:    $V_e = V_e - \{v'\}$ 
6:   update  $\mu(v)$  for neighbors  $v$  of  $v'$ 
7: end while

```

the remaining vertices in terms of their Markowitz degree. The elimination sequence follows the procedure of dynamically applied minimum Markowitz. At some point in time, a vertex whose Markowitz degree is above a certain threshold may be next in the elimination sequence. The elimination of such a vertex is considered to be too expensive.

The application of 1 : n as a preprocessing step accounts for unfortunate situations where 2 : 1 or 1 : n vertices have a high Markowitz degree, but can nevertheless be eliminated at low cost. The expected problem of selecting an appropriate threshold turns out to be simple. Due to the application of 1 : n and the manner the DAG is created, there is little room for choosing a threshold. Here $t = 6$ is used.

Example 3.2.2. Table 3.1 summarizes the assessment of eliminable vertices in terms of their Markowitz degree respectively relative Markowitz degree for the favorite example. For statically applied strategies, this assessment determines the elimination sequence.

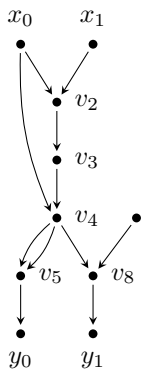


Figure 3.8:
Initial DAG.

vertex	Markowitz degree	dependence degree	relative Markowitz degree
v_2	2	4	-2
v_3	1	4	-3
v_4	6	4	2
v_5	2	2	0
v_8	2	2	0

Table 3.1: Markowitz degree and relative Markowitz degree for the DAG on the left.

If minimum Markowitz is applied statically, vertices are eliminated as follows. First, v_3 is eliminated. As v_2 , v_5 and v_8 have identical Markowitz degrees, the tiebreaker resolves the draw. For tiebreaker ‘natural’, the elimination order is v_2 , v_5 , v_8 ; tiebreaker ‘reverse’ chooses the opposite order. Last, v_4 is eliminated.

If minimum Markowitz is applied dynamically, the Markowitz degree of the remaining vertices is recomputed after each elimination. Since the elimination of a vertex can only affect direct predecessors and direct successors, the Markowitz degree of most vertices does not change. Unless explicitly stated, the values given in Table 3.1 remain valid. Using tiebreaker ‘natural’, the elimination order is v_3 , v_2 , v_5 . After the elimination of v_5 , the Markowitz degree of v_4 is updated to 4. Then v_8 and v_4 are eliminated. The elimination sequence determined with tiebreaker ‘reverse’ is v_3 , v_8 and v_5 . The elimination of v_5 updates the Markowitz degree of v_4 to 4. Next v_2 and v_4 are eliminated.

If the elimination sequence is determined according to static relative Markowitz, the vertices are eliminated in ascending order with respect to their relative Markowitz degree. Consequently, v_3 is eliminated first, followed by v_2 . The order in which the vertices v_5 and v_8 are eliminated depends on the tiebreaker. Finally, v_4 is eliminated.

Threshold Markowitz applies $1 : n$ with tiebreaker ‘reverse’ as a preprocessing step. In the process, v_3 , v_8 , v_5 and v_2 are eliminated; Example 3.3.1 demonstrates these changes step by step. The remaining vertex v_4 at this point in time has two ingoing and two outgoing edges. Thus, the Markowitz degree of v_4 is 4, as predicted by the dependence degree. As the Markowitz degree is below the threshold, v_4 is eliminated.

3.2.4 Path Count

DAGs usually contain substructures that are worthwhile being treated as black box subprograms. Structures such as subsequent $1 : 1$ vertices or funnels have obvious

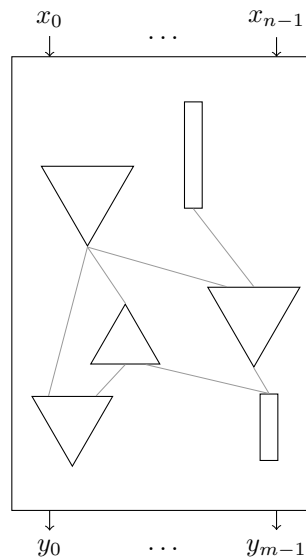


Figure 3.9: Black box view with substructures of interest.

elimination sequences, which pay for themselves. Such substructures are partially, though unintentionally, covered by the strategy $1 : n$. Something similar holds for the strategy triangles, a variant of the strategy $1 : n$. The naming of this strategy in fact attributes to the observation that triangle-shaped substructures of the DAGs are eliminated quite successfully on small-sized DAGs. Nonetheless, there is room for improvement. Consider for instance subsequent $1 : 1$ vertices. Since these vertices are prioritized in the elimination sequence, they are eliminated one after another without recognizing the substructure as a whole. If substructures slightly deviate from the perfect pattern, the strategies either cover the substructures only to a limited degree or choose a suboptimal elimination sequence.

Hence, the goal is to identify and eliminate worthwhile substructures, as illustrated in Figure 3.9. So to speak, this section generalizes the strategies $1 : n$ and triangles, respectively.

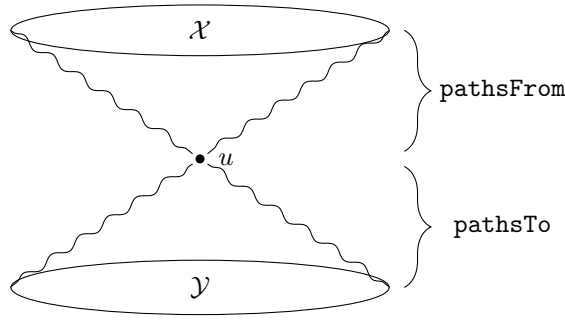
- ∇ -shaped substructures with several inputs and one output are best eliminated from bottom to top. It is likely to have such tapering triangles in real code because assignments with a combination of operations on the right-hand side create this type of structure.
- $\triangle$ -shaped substructures with one input and multiple read accesses probably occur seldom in real code. But anyway, such structures should be eliminated from top to bottom.
- Tube-shaped substructures $\sqcap$ with only a few inputs and a few outputs can be condensed into a few vertices. This structure is common since it assembles separate branches in the DAG. Note that this is a more general view than only a sequence of $1 : 1$ vertices.

In order to eliminate triangular substructures, two tasks must be accomplished. First, the tip of the triangles, or, in other words, the starting point for the elimination procedure, must be located. Second, the end of the substructure has to be detected. Similarly, the challenge for tube-shaped substructures is to identify suited boundaries.

The starting point to accomplish this task is the search for central vertices. For this purpose, the number of paths running through a vertex, here referred to as **pathCount** value, is used to assess the importance of a vertex. In contrast to the Markowitz degree, not only the immediate neighborhood accounts to a vertex, but the entire $\prec$ -relation connecting the independent and the dependent vertices.

Since the source vertices representing the independent variables and the target vertices embodying the dependent variables are always the same, **pathCount** is split up. Figure 3.10 illustrates this.

$$\text{pathCount}(u) = \left(\sum_{s \in \mathcal{X}} \text{pathsFrom}(s, u) \right) \cdot \left(\sum_{t \in \mathcal{Y}} \text{pathsTo}(u, t) \right)$$

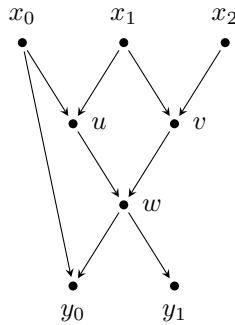
Figure 3.10: Decomposition of **pathCount** for a dynamic programming solution.

$$\text{pathsFrom}(s, u) = \begin{cases} 1 & s = u \\ \sum_i \text{pathsFrom}(s, u.\text{predecessor}_i) & u \text{ has predecessors} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{pathsTo}(u, t) = \begin{cases} 1 & u = t \\ \sum_i \text{pathsTo}(u.\text{successor}_i, t) & u \text{ has successors} \\ 0 & \text{otherwise} \end{cases}$$

Using a dynamic programming approach, intermediate results can be reused. Two look-up tables of size $|V|$ are required to tabulate the results of **pathsFrom** and **pathsTo**. The table layout $[value(v_0), value(v_1), \dots, value(v_{n-1})]$ corresponds to the order in which the vertices are created. In case of **pathsFrom**, the look-up table is filled from left to right; in case of **pathsTo**, one run through the look-up table in the opposite direction suffices. The layout ensures that required inputs, if any, are available when $value(v_i)$ is computed.

Example 3.2.3. Figure 3.11 displays a DAG with three eliminable vertices.

Figure 3.11: Sample DAG for **pathCount**.

The order of the values in the look-up vectors **pathsFrom** and **pathsTo** corresponds to the order of the vertices induced by $\prec$. Regarding **pathsFrom**, the entries associated with the independent variables are set to one. If this vector is processed from left to right, it is guaranteed that any data required for the computation is available.

The **pathsFrom**-value of a variable is equal to the sum of its predecessors' **pathsFrom** values. When for instance v is being processed, the entries of its predecessors x_1 and x_2 are accessed.

x_0	x_1	x_2	u	v	w	y_0	y_1
1	1	1	2				



Analogously, the look-up vector **pathsTo** is processed from right to left. The entries associated with the independent variables are initialized with one. Finally, the component-wise vector product yields the **pathCount** values.

	x_0	x_1	x_2	u	v	w	y_0	y_1
pathsFrom	1	1	1	2	2	4	5	4
	*							
pathsTo	3	4	2	2	2	2	1	1
	=							
pathCount	3	4	2	4	4	8	5	4

The **pathCount** values open up two options. On the one hand, one can specifically search for certain peculiarities regarding **pathCount** values. Tips of triangles should have a higher **pathCount** than vertices in close proximity. Peak values therefore indicate potential starting points for forward or reverse elimination of triangles. *Structural path count* implements this kind of search. On the other hand, the values are a means to assess how central or peripheral a vertex is. *Average path count* eliminates vertices whose **pathCount** value is below the average. The vertices are eliminated in ascending order with respect to their **pathCount** value, which hopefully eliminates the most peripheral vertices first.

Algorithm 2 gives the pseudocode for average path count. Average path count requires a tiebreaker to decide on a sequence of indices making up the elimination sequence. Vertices are processed in increasing order with respect to their **pathCount** values to eliminate unimportant parts of the DAG first. Vertices not on a path between the independent and dependent vertices exhibit a **pathCount** value of zero. Constants fall into this category. Since constants, in addition, are not touched in the elimination process, the number of constant vertices is not included in the overall vertex count used for the computation of the average value. As no activity analysis is performed, vertices which, maybe indirectly, solely depend on constant vertices can skew the assessment in the form of an average path count. Those vertices have a **pathCount** value of zero, but add to the vertex count. In terms of their **pathCount** value, those vertices are regarded to be peripheral. Case Study A.2 gives an example where this behavior occurs. Though problematic in theory, compiler optimization mitigates the problem for the most part due to constant propagation.

Algorithm 2 Average Path Count.

Input:

- DAG with vertices $V = \{v_0, v_1, \dots, v_{n-1}\}$
- set of independent vertices $V_{indep} \subseteq V$, set of dependent vertices $V_{dep} \subseteq V$
- $\text{pathCount}[] = [\text{pathCount}(v_0), \text{pathCount}(v_1), \dots, \text{pathCount}(v_{n-1})]$

```

1:  $cnt = |V| - |\{v \in V \mid v.role \equiv constant\}|$ 
2:  $avg = \left( \sum_{i=0}^{n-1} \text{pathCount}[i] \right) / cnt$ 
3: determine a sequence of indices  $i_0, i_1, \dots, i_{n-1}$  to access values in  $\text{pathCount}[]$ 
   in ascending order
4: for  $i = i_0, \dots, i_{n-1}$  do
5:   if  $\text{pathCount}[i] \leq avg \wedge v_i \notin (V_{dep} \cup V_{indep} \cup \{v \in V \mid v.role \equiv constant\})$ 
6:     eliminate  $v_i$ 
7:   else
8:     break
9:   end if
10: end for

```

Structural path count, given in Algorithm 3, harnesses **pathCount** values to identify substructures of interest. After every major change in the DAG, the **pathCount** values are recomputed. Each time the vertex with the highest **pathCount** value is chosen to serve as starting point for the next steps. If several vertices exhibit an equal **pathCount** value, the vertex with the smallest ID is chosen. This decision reflects two considerations. First, DAGs are likely to have a couple of ∇ -shaped triangles. Second, the challenge is to define the end of a triangular substructure. For this reason, ∇ -shaped and $\triangle$ -shaped triangles are treated differently. Though of interest, tube-shaped structures are not directly addressed. However, the setup for processing ∇ -shaped triangles should cover tubes to some extent.

If the DAG exhibits a $\triangle$ -shaped triangle comprising several vertices rather than a single $1 : n$ vertex with a large number of outgoing edges, there is no harm in postponing the expansion of the triangle. The procedure for ∇ does touch any of the involved vertices and thereby does not skew the structure. It is safe to select the vertices at a later stage because the smallest ID is chosen in case of a draw.

By contrast, ∇ -shaped triangles are expanded as far as possible. Naturally, central vertices exhibit high **pathCount** values. Combined with the implicit tiebreaker, the DAG is split into smaller search areas, namely the chosen vertex and its expansion upwards potentially up to the independent vertices. If, by contrast, for tied vertices the vertex with the largest ID were chosen, the expansion space would be pruned.

Vertices of type $n : m$ are obviously candidates for having high **pathCount** values, but are intentionally not used as starting points. If an $n : m$ vertex being the central vertex of an hourglass-shaped structure is encountered, either half of the structure is processed at some point.

Algorithm 3 Structural Path Count.

Input:

- DAG with vertices $V = \{v_0, v_1, \dots, v_{n-1}\}$
- set of independent vertices $V_{indep} \subseteq V$, set of dependent vertices $V_{dep} \subseteq V$
- $\text{pathCount}[] = [\text{pathCount}(v_0), \text{pathCount}(v_1), \dots, \text{pathCount}(v_{n-1})]$

```

1: done = false
2: update = true           // flag to (de)active recomputation of  $\text{pathCount}[]$ 
3:  $c = 0$                    // index of candidate vertex
4:  $c' = 0$                   // index of previous candidate vertex
5:
6: eliminate all 1 : 1 vertices
7: while  $\neg \text{done}$  do
8:   if update
9:     recompute entries in  $\text{pathCount}[]$ 
10:    for each  $v_i \in (V_{dep} \cup V_{indep})$  do
11:       $\text{pathCount}[i] = 0$ 
12:    end for
13:  end if
14:   $c' = c$ 
15:  determine the smallest index  $c$  such that
     $\text{pathCount}[c] = \max\{\text{pathCount}[i] \mid 0 \leq i \leq n-1\}$ 
16:  if  $c \equiv c' \vee \text{pathCount}[c] \equiv 0$ 
17:    done = true
18:  end if
19:  switch  $v_c.type$  do
20:    case 2 : 1, 1 : 1      // eliminate  $\nabla$  from bottom to top
21:      let queue be an empty queue
22:      queue.push( $c$ )
23:      while  $\neg \text{queue.empty}()$  do
24:         $i = \text{queue.pop}()$ 
25:        for each  $v_j \in v_i.predecessors$  do
26:          queue.push( $j$ )
27:        end for
28:        if  $(v_i.type \equiv 2 : 1 \vee v_i.type \equiv 1 : 1) \wedge v_i \notin (V_{dep} \cup V_{indep})$ 
29:          eliminate  $v_i$ 
30:        end if
31:      end while
32:      update = true
33:    case 1 :  $n$            //  $\triangle$ , eliminate directly
34:      eliminate  $v_c$ 
35:      update = true
36:    case  $n : m$           // demand new candidate
37:       $\text{pathCount}[c] = 0$ 
38:      update = false
39:  end while

```

3.3 Code Generation

This section describes how the DAG is used to generate source code. The DAG is only used to maintain information about data dependencies. In other words, only the existence of edges is stored, but not their labeling. Any other information, most importantly local partial derivatives and the accumulation order, is directly written to a file. For this purpose, two types of variables are in use. On the one hand, there are two variables for each vertex. The variable $v_i \equiv \mathbf{v_i}$ holds an intermediate result of the original function; the adjoint variable $\mathbf{a_i}$ holds the adjoint value of $\mathbf{v_i}$. On the other hand, there are variables $c_i \equiv \mathbf{c_i}$ for the edges. Every edge in the original DAG is represented by such a variable and is initialized with the value of the local partial derivative. Further variables are created in the elimination process and constitute a composition of edge variables.

The generated source files implement the following five steps. An illustration of the procedure can be found in Example 3.3.1.

1. **Declaration of vertex variables and adjoints.** A vertex in the DAG corresponds to an intermediate result. For each vertex, the generated code contains two variables, namely a variable `double v_i = 0.0` for the value and a variable `double a_i = 0.0` for the adjoint value.
2. **Forward run.** The first lines copy the values of the input vector $\mathbf{x}$ to the corresponding independent vertex variables, $\mathbf{v_i} = \mathbf{x[i]}$. Based on the internal representation in the form of the DAG, the implementation of the original function is written to the source file in single-assignment form.
3. **Local partial derivatives.** A variable `double c_i` is declared for every edge in the original DAG and initialized with the value of the local partial derivative. So far the generated source files are identical regardless of the elimination strategy.
4. **Implementation of the elimination sequence.** The elimination of a vertex can either create a new edge or increment the label of an existing edge. A new edge creates a fresh variable whose value equals the product of the corresponding edge variables. In the event of absorption, the value of an existing edge variable is incremented.
5. **Seeding, reverse run and harvesting.** The seed for $\mathbf{y}_{(1)}$ is passed via `a1_y`; values are copied to the adjoint variables of the dependent vertices. The edges and vertices left in the DAG after the elimination procedure determine the reverse run. The edge labels are multiplied along paths and increment the respective adjoint variables. Eventually, the adjoint values of the independent variables $\mathbf{x}_{(1)}$ are written back to `a1_x`.

The resulting source files comprise large blocks of static code. As there are neither loops nor branches, the control flow is clear. Hence, there are no obstacles for the compiler regarding code optimization.

Example 3.3.1. This example demonstrates the code generation with the favorite example. The elimination sequence is chosen according to $1 : n$ with tiebreaker ‘reverse’. Figure 3.12 shows the DAG that is the starting point for the elimination procedure. The elimination sequence is v_3, v_8, v_5, v_2 . Figure 3.13 illustrates in a step-by-step fashion how the elimination sequence is processed, i.e. how the DAG is (internally) updated and what code is written to the source file. The entire source code is given in Listing 3.1 at the end of the example.

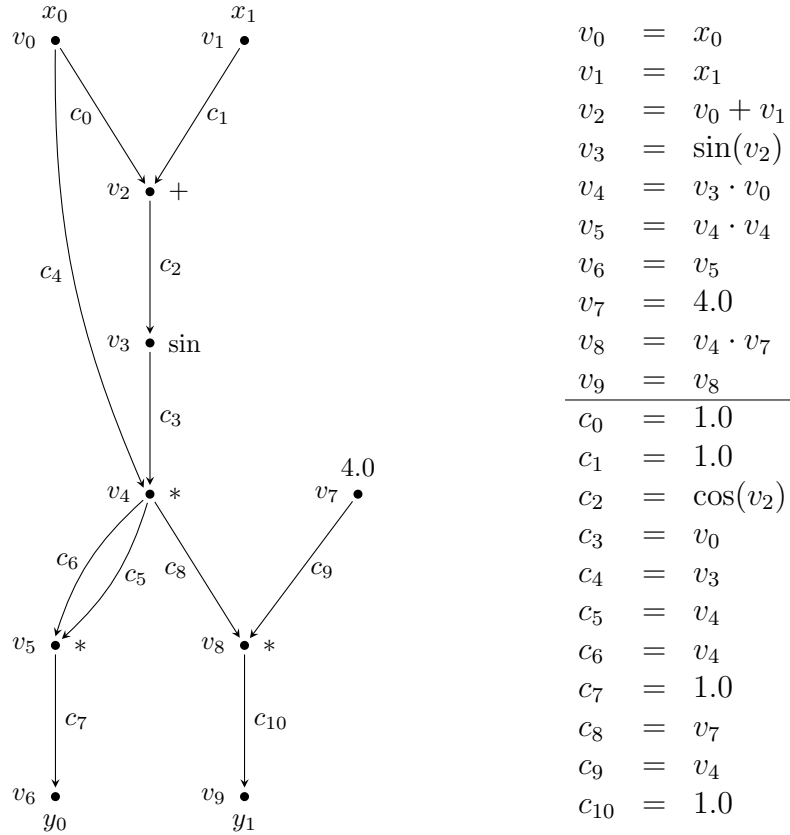
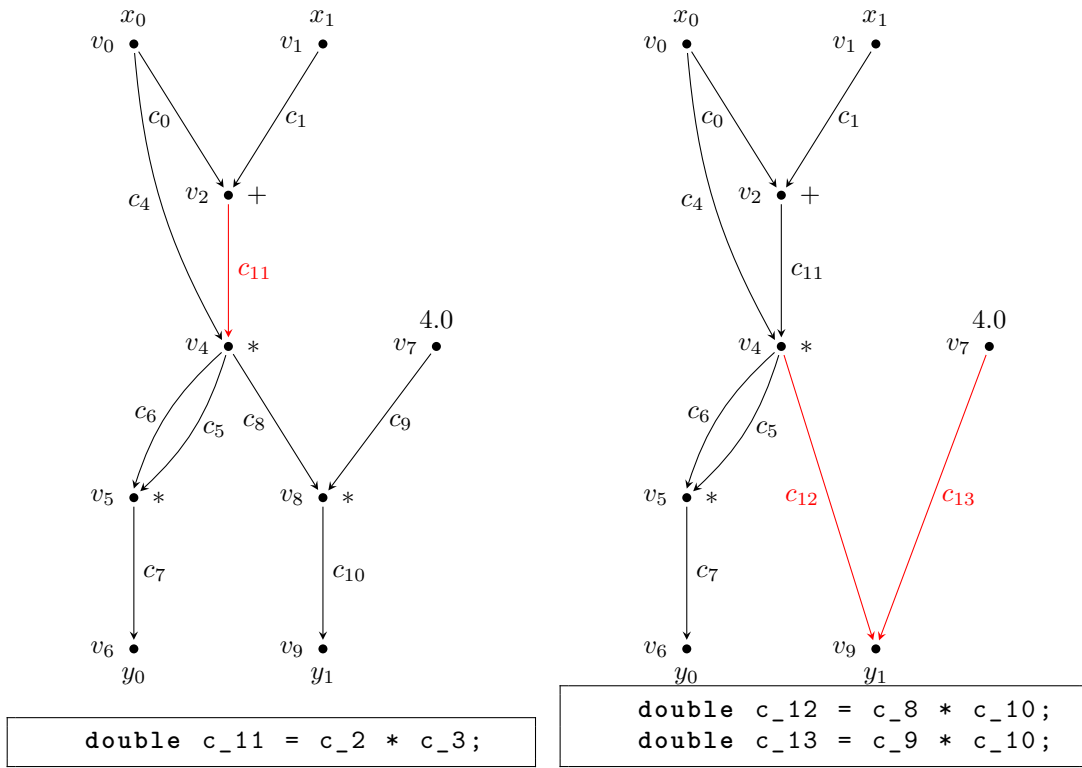


Figure 3.12: DAG for the favorite example as built up by overloading.

Some elimination strategies intentionally do not eliminate vertices with multiple ingoing and multiple outgoing edges to save multiplications. This also applies to v_4 in this example. Since v_4 remains in the DAG, the code written to the source file implements

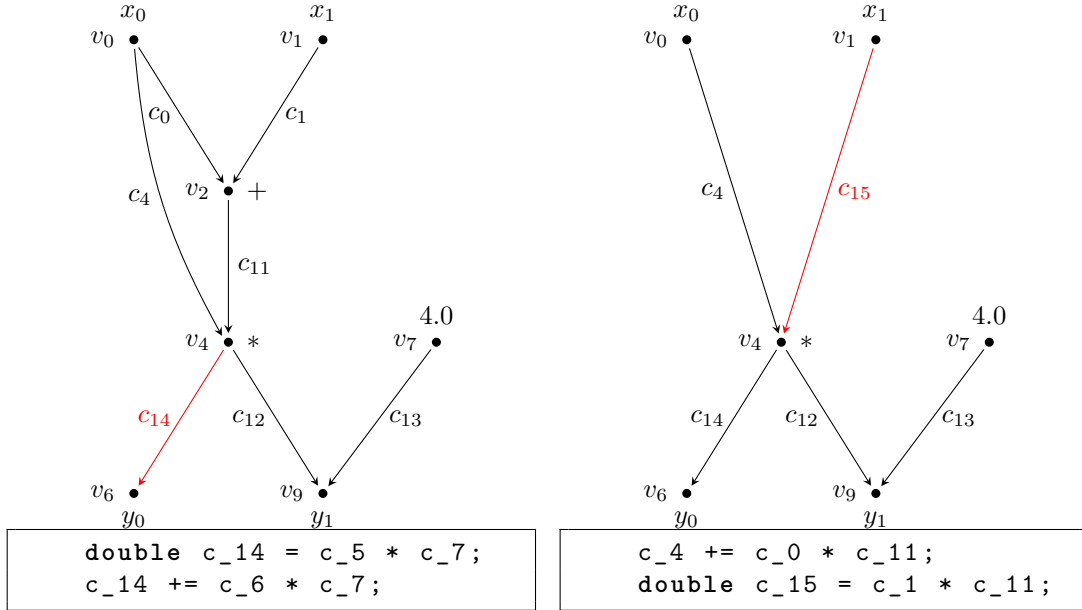
$$\begin{aligned}
 \underbrace{\begin{pmatrix} a1_x[0] & a1_x[1] \end{pmatrix}}_{\mathbf{x}_{(1)}^T} &= \left(\underbrace{\begin{pmatrix} a1_y[0] & a1_y[1] \end{pmatrix}}_{\mathbf{y}_{(1)}^T} \cdot \begin{pmatrix} c_{14} \\ c_{12} \end{pmatrix} \right) \cdot \begin{pmatrix} c_4 & c_{15} \end{pmatrix} \\
 &= \mathbf{a_4} \cdot \begin{pmatrix} c_4 & c_{15} \end{pmatrix}
 \end{aligned}$$

with $\mathbf{a_4}$ being the adjoint variable of v_4 . The edge variables c_{12} , c_{14} , c_4 and c_{15} can be traced to compositions of local partial derivatives, whose bracketing and accumulation scheme is prescribed by the elimination sequence.



(a) DAG after the elimination of v_3 . The elimination has created one new edge, namely c_{11} .

(b) Next, v_8 is eliminated. In doing so, two new edges, c_{12} and c_{13} , are created.



(c) The elimination of v_5 creates only one new edge. This edge absorbs what is generated from $c_6 \cdot c_7$.

(d) The elimination of v_2 creates one new edge c_{15} . Due to absorption, $c_0 \cdot c_{11}$ augments c_4 .

Figure 3.13: Elimination sequence conducted by the strategy 1 : n with tiebreaker ‘reverse’. One eliminable vertex (v_4) is left over.

```

1 void a1_f(vector<double>&x, vector<double>&a1_x, vector<double>&
    a1_y){
2     double v_0 = 0.0; double a_0 = 0.0; }
3     ... } vertex variables and adjoints
4     double v_9 = 0.0; double a_9 = 0.0; }
5     v_0 = x[0];
6     v_1 = x[1];
7     v_2= v_0+v_1;
8     v_3=std::sin(v_2);
9     v_4= v_3*v_0;
10    v_5= v_4*v_4;
11    v_6=(v_5);
12    v_7=4;
13    v_8= v_4*v_7;
14    v_9=(v_8);
15    double c_0 = 1.0;
16    double c_1 = 1.0;
17    double c_2 = cos(v_2);
18    double c_3 = v_0;
19    double c_4 = v_3;
20    double c_5 = v_4;
21    double c_6 = v_4;
22    double c_7 = 1.0;
23    double c_8 = v_7;
24    double c_9 = v_4;
25    double c_10 = 1.0;
26    double c_11 = c_2 * c_3;
27    double c_12 = c_8 * c_10;
28    double c_13 = c_9 * c_10;
29    double c_14 = c_5 * c_7;
30    c_14+= c_6 * c_7;
31    c_4+= c_0 * c_11;
32    double c_15 = c_1 * c_11;
33
34    a_6 = a1_y[0];
35    a_9 = a1_y[1];
36    a_4+= c_12 * a_9;
37    a_4+= c_14 * a_6;
38    a_1+= c_15 * a_4;
39    a_0+= c_4 * a_4;
40    a1_x[0]=a_0;
41    a1_x[1]=a_1;
42 }

```

forward run
 initial local partial derivatives
 elimination sequence
 seed
 reverse run
 harvest

Listing 3.1: Code generated with elimination strategy 1 : n and tiebreaker ‘reverse’ for the favorite example. Note that the function value $\mathbf{y} = f(x_0, x_1)$ is not written back.

The generated source codes typically contain a lot of unused variables and maybe even dead code, computations whose result is never read. Moreover, the codes exhibit trivial computations such as multiplications with ± 1 . Flaws of this sort are removed by the compiler. Section 3.5 takes up on this example and illuminates some more traits of compiler optimization.

3.4 Helmholtz Example

Inspired by [13], the next case study covers the Helmholtz energy function with $f: \mathbb{R}^n \rightarrow \mathbb{R}$. According to [13] referencing [19], the energy function is used for modeling mixed fluids in phase equilibria. The particular form used here describes the energy density of reservoir oils. The original implementation available from [29] is part of the test functions for ADOL-C and has been rewritten for each interface.

For the runtime tests the dimension was set to $n = 200$. The DAG that is built up has 200 independent vertices, 1 dependent vertex and 82830 intermediate vertices,

strategy and tiebreaker	source file size	object file size	runtime
natural order	363410355 B	?	?
reverse order	12376539 B	2355760 B	1.45 s
static minimum Markowitz			
- random	31740129 B	2333384 B	1.49 s
- natural	192200772 B	2345760 B	1.51 s
- reverse	12384470 B	2355760 B	1.50 s
dynamic minimum Markowitz			
- natural	19608841 B	2347176 B	1.50 s
- reverse	12368070 B	2355760 B	1.49 s
static relative Markowitz			
- random	12380749 B	2355216 B	1.51 s
- natural	12393406 B	2369712 B	1.50 s
- reverse	12368111 B	2354176 B	1.45 s
1 : n			
- natural	192407105 B	2433448 B	1.53 s
- reverse	12368070 B	2355760 B	1.45 s
triangles	12375435 B	2356592 B	1.48 s
structural path count	12368276 B	2352112 B	1.50 s
average path count			
- random	92652312 B	2368432 B	1.46 s
- natural	118275130 B	2377736 B	1.49 s
- reverse	66332188 B	2373208 B	1.47 s
dco/c++			*2.32 s
dcc			5.50 s

Table 3.2: Runtimes of different strategies for the Helmholtz function in comparison with the performance of `dco/c++` and `dcc`¹, respectively.

¹Source code transformation tool for C++ code, see [8] for a description.

of which 62119 are eliminable. Timings are based on 10000 evaluations of the gradient and an average of 10 runs of the executable. In case of `dco/c++`, the timing does not include the recording in the tape, but is restricted to the interpretation of the tape. The rationale for excluding the recording is to account for the advantage of `dco/c++` over adjoint code produced by the code generator. The tape can be interpreted with different seeds, for example when the entire Jacobian matrix shall be computed. Consequently, it suffices to record the forward run once. The generated adjoint codes, by contrast, must execute the forward run in any case because intermediate results cannot be stored. Table 3.2 lists the results, which have been measured on the following system.

compiler:	<code>g++ 4.7.2</code>
compiler flags:	<code>-O4 -Ofast</code>
CPU:	Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30 GHz
L1 cache:	6×32 kB instruction cache 6×32 kB data cache
L2 cache:	6×256 kB
L3 cache:	15 MB
RAM:	126 GB

The row with natural order is left open, because the compilation time rises quickly when the size of the source files increases. See Case Study A.1 to get an idea about the relation between source file sizes and compilation time.

Telling by the variation in the source file sizes, the elimination strategies perform markedly different on the Helmholtz function. Differences in the file sizes can, for the most part, be traced back to the implementation of the elimination sequence. Thus, large file sizes result from additional edges and thereby also additional multiplications created in the elimination process. Moreover, the DAG seemingly consists of large groups of tied vertices. Though the exact number of tied vertices undoubtedly depends on the specific strategy, the tiebreaker has significant impact on each elimination strategy. Based on the multiplication count in the file sizes, the expected outcome is that strategies yielding small file sizes generally perform better.

Although the file size of the adjoint source codes varies substantially, the size of the object files is in the same order of magnitude, around 2.3 MB. Similarly, the runtime measurements show rather similar results. The elimination strategy has little effect on the overall runtime; maybe there is a tendency to favor elimination in reversed order. Thus, the expected superiority of certain strategies does not manifest itself in this example. Instead, compiler optimization essentially equalizes the source files. In this sense, the choice of the elimination strategies is negligible. Even the theoretical advantage in terms of the multiplication count does not express itself. Table 3.3 lists the number of multiplication instructions as set by the compiler. As neither the source files nor the object files exhibit any branches or jumps, it suffices to count the occurrences of the respective machine instructions in the disassembly. The multiplication counts abstract from the exact FPU instruction (for example `fmul`, `fmulp` or `fimul`) and subsumes any machine instruction related to floating-point multiplications under the multiplication count. Apparently, the theoretical

strategy and tiebreaker	multiplication count
reverse order	42053
static minimum Markowitz	
- natural	42163
- reverse	42053
dynamic minimum Markowitz	
- natural	41692
- reverse	42053
1 : n	
- natural	43607
- reverse	42053
triangles	41862
structural path count	42180
average path count	
- natural	41483
- reverse	41483

Table 3.3: Selection of multiplication counts for different strategies for the Helmholtz function using $n = 200$. The multiplication count corresponds the number of respective machine instructions.

multiplication count being minimized by means of vertex elimination tells little about the effective multiplication count in the object files. The next section examines reason why the choice of the elimination strategy does not have a significant impact on the runtime.

3.5 Case Study: Effectiveness of Compiler Optimization

The intention of this section is to discuss reasons why the superiority or inferiority of strategies does not manifest itself in terms of the overall runtime. This should also give some ideas of why the codes generated by vertex elimination AD outperform dcc and dco/c++. For this purpose, reconsider the favorite example.

1	0:	8b 44 24 04	mov	0x4(%esp),%eax
2	4:	8b 00	mov	(%eax),%eax
3	6:	dd 00	fldl	(%eax)
4	8:	dd 40 08	fldl	0x8(%eax)
5	b:	d8 c1	fadd	%st(1),%st
6	d:	8b 44 24 10	mov	0x10(%esp),%eax
7	11:	d9 fb	fsincos	
8	13:	8b 00	mov	(%eax),%eax
9	15:	d8 ca	fmul	%st(2),%st

10	17:	d9 ca	fxch	%st(2)
11	19:	d8 c0	fadd	%st(0),%st
12	1b:	dc 08	fmull	(%eax)
13	1d:	d8 c9	fmul	%st(1),%st
14	1f:	d9 05 00 00 00 00	flds	0x0
15	25:	dc 48 08	fmull	0x8(%eax)
16	28:	8b 44 24 08	mov	0x8(%esp),%eax
17	2c:	8b 00	mov	(%eax),%eax
18	2e:	de c1	faddp	%st,%st(1)
19	30:	d9 c9	fxch	%st(1)
20	32:	d8 c2	fadd	%st(2),%st
21	34:	d8 c9	fmul	%st(1),%st
22	36:	dd 18	fstpl	(%eax)
23	38:	de c9	fmulp	%st,%st(1)
24	3a:	dd 58 08	fstpl	0x8(%eax)
25	3d:	c3	ret	

Listing 3.2: Disassembly of gcc-generated code in Intel syntax for the favorite example with elimination strategy 1 : n and tiebreaker ‘reverse’. Optimization has been set to `-O4 -Ofast`.

Listing 3.2 shows the disassembly of the code of the favorite example. The target platform is an Intel Core i7 with an eight floating-point register stack consisting of $\%st \equiv \%st(0), \dots, \%st(7)$. Technical data on the Intel Core i7 is taken from [14], see also Figure 3.14. Though this example is small, it reveals effects that strengthen with increasing code size.

- **Constant propagation and constant folding.** Constant values are propagated as far forward as possible. The compiler removes constants that correspond to constant vertices (such as v_7 in the example) and constant partial derivatives.
- **Padding.** The compiler inserts no-operation instructions to advance the instruction pointer. In this particular example there is the 2-byte no-operation `mov (%eax), %eax`. The purpose of no-operations is the alignment of instructions.
- **Lack of branch mispredictions.** Since the code is static and does not contain any conditional jumps, there is no branch prediction. Thus, register states can be updated immediately as opposed to waiting until the instruction is no longer speculative. Mispredictions and incorrect speculation are especially expensive: On the one hand, branch mispredictions cost 15 cycles and thereby are more expensive than an L1 cache miss. On the other hand, speculative instructions that are eventually annulled fill the pipeline at the expense of other instructions. This causes idle cycles while the correct instructions are being fetched, decoded and inserted into the pipeline structure. Hence, the program benefits from better instruction-level parallelism.
- **Reordering of instructions.** The disassembly reveals that the compiler heavily restructures the program. Reordering enables more effective hardware exploitation including enhanced instruction-level parallelism, data-cache locality and register allocation. The disassembly shows that indeed dependencies of

type	time
L1 cache miss	~ 10 cycles
L2 cache miss	~ 30 cycles
L3 cache miss	$\sim 130 - 135$ cycles

Table 3.4: Costs of cache misses for the i7, taken from [14].

instructions are exploited to improve the utilization of the register stack – an effect that becomes more important with increasing code size. Values should preferably be stored in registers because accessing data in registers is much faster than accessing data from memory. Once present in the floating-point registers, values are used in all operations requiring the value and are then overwritten. As a consequence, the floating-point register stack is kept small. The task of storing values in registers as long as possible is alleviated through the program structure with independent branches in the DAG.

- **SSE.** Due to reordering of instructions, the code benefits from SSE. Special instructions such as the computation of intrinsic functions work on short fixed-size arrays. The disassembly exhibits an example of this behavior: `fsincos` computes two values at the same time. The execution of `fsincos` is faster than the successive execution of `fsin` and `fcos`.
- **Interleaving forward sweep and reverse sweep.** There is no strict separation between the computation of the values of vertex variables and the values of partial derivative variables. The value of a vertex variable must be kept until all evaluations requiring this value have been executed. If the value of a vertex variable contributes to only a few computations, prioritizing those computations reduces the set of values which must be stored. This includes statements of the reverse sweep, e.g. `0x19` in the example overwrites a value that is no longer needed. SSE promotes this behavior.

The quality of the code heavily depends on the optimization level. The optimization parameters have impact on the multiplication count, the memory requirements and the usage of SSE. When the favorite example is compiled with `-O2`, the number of instruction grows by a factor of 1.6, mostly from `mov` instructions. In comparison with the previous settings, the register allocation is poor. The floating-point register stack grows up to five entries and four general registers are used (compared to one for `-O4 -Ofast`). Furthermore, it depends on the optimization level to what extent SSE is used. In this example, the usage of `fsincos` requires an optimization level of no less than `-O3`.

The findings of this small example are in accord with the results of the Helmholtz example. Although the size of the source files differ significantly, the size of the object files is similar. For some strategies the compiler even generates identical object files. The accumulation order prescribed by some elimination strategy is at best a recommendation. The compiler reorders instructions freely to exploit

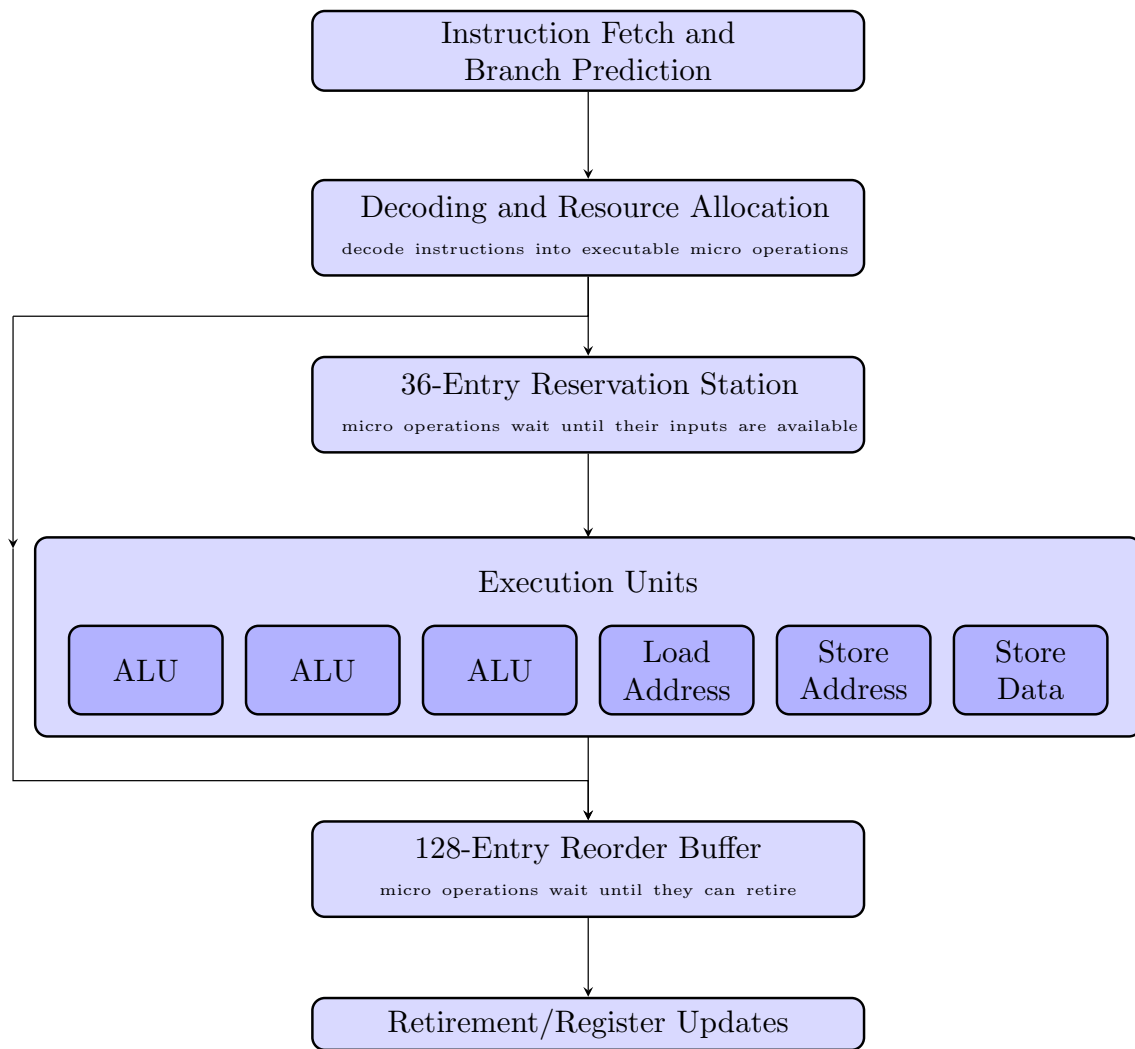


Figure 3.14: Pipeline structure of the Intel Core i7, adapted from [14] and simplified. There are six execution units, three of which are for arithmetic operations. Hence, up to three arithmetic operations can be executed in parallel.

hardware features. As side benefit, compiler optimization obviously alleviates the dependence of the program performance on the way the program is written.

Recall the motivation for employing different elimination strategies. The elimination of vertices ought to be an instrument for minimizing the number of multiplications. Indeed, different strategies generate codes with distinct multiplication counts. Compiler optimization, however, equalizes the codes for the most part. Although some disparity in the multiplication count remains, this disparity does not have a major impact on the overall runtime. Considering that several ALUs (three in Figure 3.14) allow for parallel execution of floating-point operations, the multiplication count may not be that crucial. This raises the question of whether a simple cost model such as the multiplication count is still appropriate on modern superscalar processors. Instead, perhaps the memory access pattern is the more important factor, in

the sense of the reducing read and write accesses to registers and caches. Regarding this, Table 3.4 lists how costly cache misses are.

As stated in Chapter 2, the advantage of vertex elimination over forward and reverse mode is that virtually any accumulation sequences can be chosen, which, in turn, yields a reduction of the floating-point operation count. At least for the examples tested here, this statement must be qualified. The advantage does not come from a reduction of the floating-point operation count, but from the fact that the code is static and therefore benefits from superior compiler optimization. This is also the reason why the codes outperform `dcc` and `dco/c++`, respectively.

Tadjouddine et al. [27] come to similar results, though following an entirely different track. Their work evaluates the potential trade-off number of memory accesses versus reduction of floating-point operations for their vertex elimination algorithm. The two parameters have been varied through different accumulation strategies or manual reordering of statements of the generated Jacobian code. Their test series includes several architecture types and tests with various levels of compiler optimization. They, too, conclude that the number of load and store operations has a profound impact on the overall performance. They reason that the increasing gap between CPU performance and memory performance promotes programs with low memory traffic. In particular, they remark that a small reduction of the floating-point operation count does not per se result in an improved runtime – an observation which also applies to e.g. the Helmholtz example. However, their test series also shows that for data present in registers, floating-point operations dominate the runtime. Hence, provided that the data is available in the registers, the performance can be improved even further with reducing the number of floating-point operations.

At least in the papers [27, 28, 10], Tadjouddine et al. do not attribute the performance boost they measure to benefits coming from static code or hardware features. In the tests run here, it turns out that especially SSE is a driving force for reordering instructions. Case Study A.3 takes up on this subject and illustrates the impact of compiler flags.

As opposed to the results presented here, Tadjouddine et al. [28] observe in their test series that their code produced by vertex elimination falls behind in terms of runtime on some compiler/platform combinations. The decisive factor is whether or not the processor supports out-of-order execution. Processors maintain a queue (in Figure 3.14 this is called reservation station) where instructions wait until their inputs are available. If an instruction whose inputs are not yet available is next in line, processors supporting out-of-order execution can prepone other instructions in the queue whose inputs are already available. Otherwise, the processor waits until the inputs of the instruction next in line are available, which introduces idle time. If the processor does not support out-of-order execution, it is essential that the compiler be capable of instruction scheduling in order to improve instruction-level parallelism. Despite their code being static, the compiler does not accomplish reordering of instructions. However, manual reordering of statements in the derivative code solves the performance issues. This result dates back to 2002 and raises the question if compilers still lack maturity on certain platforms. If so, the choice of the elimination strategy would indeed make a difference.

3.6 Integration into dco/c++

Recall that the ultimate goal of the code generator is the outsourcing of expensive basic blocks whilst the remainder of the program is differentiated with `dco/c++`. Thus, a function

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &= \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \end{aligned}$$

has a computationally expensive basic block $\hat{\mathbf{y}} = f(\hat{\mathbf{x}})$ whose adjoint computation is replaced by the code generator's output.

The external function interface of `dco/c++` [2, 15] allows for the integration of an external function f and its corresponding external adjoint function, here called adjoint f . In this particular case adjoint f is implemented by one of the optimized adjoint codes from the code generator. From the perspective of `dco/c++`, the function is a black box function, whose inputs and outputs are registered. When f is called, `dco/c++` does not record any intermediate results in the tape. In the reverse run, in comparison to the basic reverse mode of `dco/c++`, the basic block is superseded with an optimized adjoint version. Thus, adjoint f implements

$$\begin{aligned} \hat{\mathbf{y}} &= f(\hat{\mathbf{x}}) && \text{(recomputation, code generator)} \\ \hat{\mathbf{x}}_{(1)} &= \nabla f(\hat{\mathbf{x}})^T \cdot \hat{\mathbf{y}}_{(1)} && \text{(local adjoints, code generator)} \\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \hat{\mathbf{x}}_{(1)} && \text{(write back results to dco/c++)} \end{aligned}$$

Figure 3.15 illustrates how `dco/c++`, f and adjoint f interact.

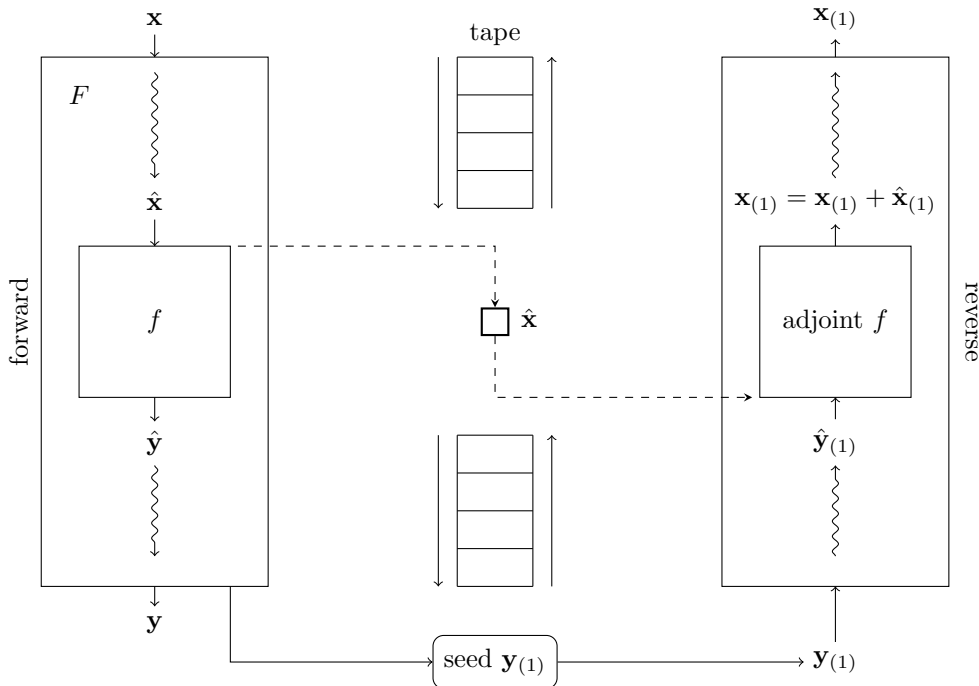


Figure 3.15: Integration of adjoint codes from the code generator into `dco/c++`. The block f is differentiated via the code generator. Everything aside from f is differentiated with `dco/c++`.

For reference, some more profound examples of how to use the external function interface and pictures for visualization are given in [24].

The basic block f obviously is required at two stages, which makes it attractive to declare f as template function. On the one hand, f is required for the generation of corresponding adjoint code. On the other hand, f must be executed in the forward run, preferably by calling the template function with an appropriate primitive data type.

The function adjoint f as given in Figure 3.15 works as an adapter between *dco/c++* and the interface of the optimized adjoint codes from the code generator

```
void a1_f(vector<double> &x, vector<double> &a1_x, vector<double> &a1_y)
```

where

- $\mathbf{x}$ is equal to $\hat{\mathbf{x}}$. The generated adjoint codes require the correct input value $\hat{\mathbf{x}}$ for the local recomputation of all intermediate values. Therefore, $\hat{\mathbf{x}}$ must be retrievable e.g. as part of checkpoint data, which is illustrated in Figure 3.15. Whenever f is executed several times with distinct values of $\hat{\mathbf{x}}$, the values obviously must be written to the checkpoint at each time. This scenario occurs, for instance, when f is part of a loop body.
- $\mathbf{a1_y}$ corresponds to $\hat{\mathbf{y}}_{(1)}$ and contains the seed values for $\mathbf{a1_f}()$. The values are fetched from *dco/c++* in a value-by-value fashion and arranged as vector to match the function signature.
- $\mathbf{a1_x}$ is a vector containing the computed adjoints, i.e. $\hat{\mathbf{x}}_{(1)}$, after calling $\mathbf{a1_f}()$. These values, in turn, must be passed component-wise to *dco/c++* to continue with the interpretation of the tape.

Thus, adjoint f creates and initializes the respective vectors and calls $\mathbf{a1_f}()$. The function $\mathbf{a1_f}()$ implements the recomputation of the forward run $\hat{\mathbf{y}} = f(\hat{\mathbf{x}})$ and the accumulation of the adjoints $\hat{\mathbf{x}}_{(1)} = \nabla f(\hat{\mathbf{x}})^T \cdot \hat{\mathbf{y}}_{(1)}$. The results $\hat{\mathbf{x}}_{(1)}$ are returned to *dco/c++* meaning that the respective adjoints of the global function F are incremented, i.e. $\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \hat{\mathbf{x}}_{(1)}$. A more code-centric illustration is part of the user documentation in Appendix B.

Due to the setup of the interfaces, overhead is created. In the forward run, copies are created in order to call f with type *double*. In the reverse run, overhead is created by arranging the values of *dco/c++* data types as vectors and vice versa. If values, say iteration-dependent constants, shall be passed in addition to $\hat{\mathbf{x}}$, even more copying is necessary. The signature of $\mathbf{a1_f}()$ requires those values and $\hat{\mathbf{x}}$ to be glued together yielding one single vector.

The remainder of this section exemplarily evaluates the trade-off copy overhead vs. speedup coming from compiler optimization. For this purpose, the CG method given in Algorithm 4 is interpreted as the function $F(\mathbf{A}, \mathbf{b}) = \mathbf{x}^*$, $\mathbf{A}\mathbf{x}^* = \mathbf{b}$ with $\mathbf{A}_{(1)}$ being aimed at [23]. The case study incorporates three variants.

Algorithm 4 Conjugate Gradient Method for solving $A\mathbf{x} = \mathbf{b}$, adapted from [18]

Input:

- starting point $\mathbf{x}_0 \in \mathbb{R}^n$
- symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$
- $\mathbf{b} \in \mathbb{R}^n$

Output:

- $\mathbf{x}^*$ with $A\mathbf{x}^* = \mathbf{b}$

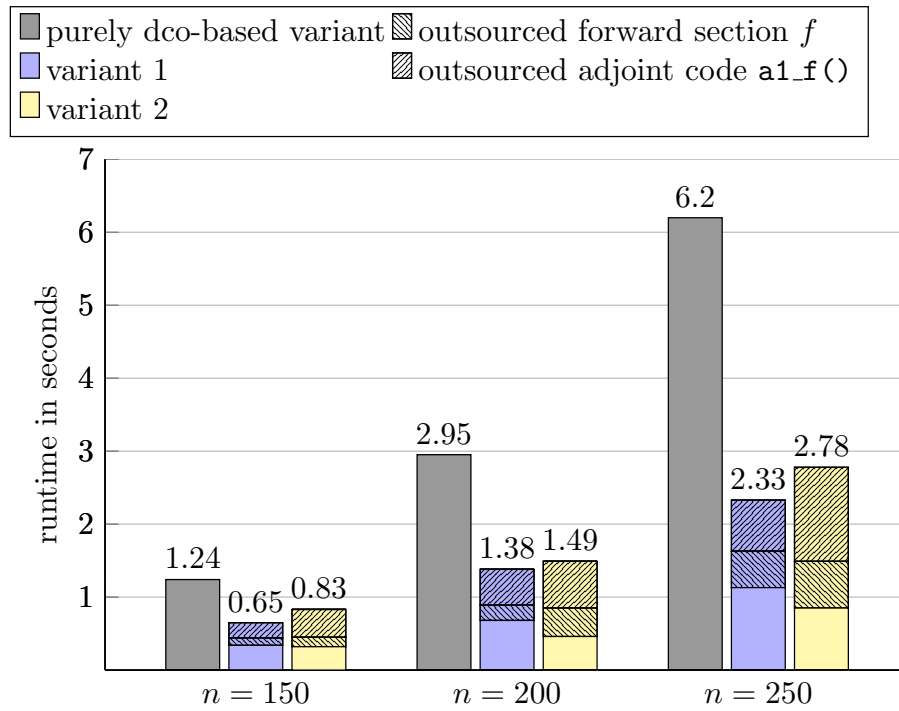
```

1:  $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ 
2:  $\mathbf{p}_0 = \mathbf{r}_0$ 
3:  $\alpha_0 = \|\mathbf{r}_0\|_2^2$ 
4: for  $m = 0, 1, \dots$  do
5:   if  $\alpha_m < \epsilon$ 
6:     return  $\mathbf{x}_m$ 
7:   end if
8:    $\mathbf{v}_m = A\mathbf{p}_m$ 
9:    $\lambda_m = \frac{\alpha_m}{\mathbf{v}_m^T \mathbf{p}_m}$ 
10:   $\mathbf{r}_{m+1} = \mathbf{r}_m - \lambda_m \mathbf{v}_m$ 
11:   $\mathbf{x}_{m+1} = \mathbf{x}_m + \lambda_m \mathbf{p}_m$ 
12:   $\alpha_{m+1} = \|\mathbf{r}_{m+1}\|_2^2$ 
13:   $\mathbf{p}_{m+1} = \mathbf{r}_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} \mathbf{p}_m$ 
14: end for

```

- **Purely dco/c++-based variant.** The computation solely relies on dco/c++ data types. Thus, there is no copy overhead.
- **Variant 1.** The matrix-vector product (line 8) has been outsourced. In order to match the interface of `a1_f()`, the code segment cut out corresponds to the function $\mathbb{R}^{n \cdot (n+1)/2 + n} \rightarrow \mathbb{R}^n$, $f(A, \mathbf{p}_m) = \mathbf{v}_m$.
- **Variant 2.** The matrix-vector product computing $\mathbf{v}_m$ and all computations using $\mathbf{v}_m$ (lines 8–10) are treated as black box function $\mathbb{R}^{n \cdot (n+1)/2 + n + 1 + n} \rightarrow \mathbb{R}^{n+1}$, $f(A, \mathbf{p}_m, \alpha_m, \mathbf{r}_m) = (\mathbf{r}_{m+1}, \lambda_m)$, which is cut out. The downside is that the copy overhead is increased.

Another option for outsourcing some computation would be to cut out a couple of subsequent iterations and treat those as one single function. The first few iterations could be used to determine an approximation of $\mathbf{x}^*$ without checking the termination criterion. Afterwards, the result could be used by dco/c++ as starting point. In this section, the termination criterion can be evaluated. In the reverse run, the final result computed by means of dco/c++ could be passed as input to `a1_f()`, which computes the overall result. Thus, the computation by dco/c++ and the outsourced code would be glued together rather than interleaved. In contrast to the variants sketched above, the overhead would be minimal. However, at least for this case study this idea is not an option because the code sizes grow quickly.

Figure 3.16: CG method performing on M_n .

A slightly modified version of the CG method is employed in the case study. In order to obtain reasonable results, the actual termination criterion is omitted. Instead, the loop is executed $10 \cdot n$ times. To be able to scale the dimension easily, the symmetric positive definite matrix

$$M_n = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

is used. The symmetry of the matrix is exploited, but the sparsity is not. The adjoint code for the outsourced segments is generated using structural path count in each case. Measurements refer to the same system as the Helmholtz example and are based on an average of 10 runs of the executable; Figure 3.16 summarizes the results.

Despite the overhead introduced by the external function interface, both variant 1 and variant 2 outperform *dco/c++*. If the extension of the outsourced code segment in variant 2 has any positive effect on the runtime compared to *dco/c++*, this speedup cannot compensate the copy overhead.

Comparing the time spent on the outsourced forward section, namely f , and the corresponding outsourced adjoint code section, namely $\mathbf{a1_f}()$, the computation of $\mathbf{a1_f}()$ is surprisingly cheap. Recall that $\mathbf{a1_f}()$ recomputes all intermediate values of f and therefore actually incorporates the cost of f .

Although the speedup in either variant looks promising at first glance, there are some problems. First, the chart gives an idea of how the runtimes evolve for smaller n . A relatively big chunk of the program must be outsourced in order to make the approach pay off. Second, the scalability of this approach is limited. The compilation time, for instance, increases by a factor of eight in both variants when scaling n from 150 to 250.

4

Conclusion

This thesis has presented a code generator for C++ implementing the vertex elimination AD approach. The code generator, in conjunction with compiler optimization, produces efficient first-order adjoint code. Given a program which is differentiated via AD by operator overloading, those first-order adjoint codes can be used to supersede the adjoint computation of static code segments. As shown in the CG example in Section 3.6, the initial investment in the form of vertex elimination coupled with compiler optimization can pay off and yield more efficient programs.

In contrast to conventional forward or reverse mode of AD, vertex elimination is not tied to monotonic orderings. Elimination strategies harness this freedom to find elimination sequences that minimize the number of floating-point multiplications. The code generator employs a variety of elimination strategies, whose performance differs significantly with respect to the theoretical multiplication count.

The codes produced by the code generator do not exhibit branches, loops or any other control structures. Instead, the codes consist of large blocks of static code. Due to the simple structure of the generated codes, compiler optimization works especially well. The compiler's primary optimization objective seemingly is the improvement of memory accesses. As a side effect, differences in the source files coming from the application of distinct elimination strategies are mostly equalized. Thus, except for very specific program structures, the choice of the elimination strategy takes a back seat to compiler optimization.

As revealed in the CG example, the coupling of the offline optimization code generator and AD by operator overloading can yield speedup. Unfortunately, the outsourcing of code segments is a delicate matter. On the one hand, the outsourced code segments must exceed a certain length in order to pay off. On the other hand, the offline optimization approach lacks scalability because the code generator produces straight-line code. The point at which the files are simply too large for the compiler to digest may be reached rather quickly.

In the current setup, the application of the code generator is very limited. Control structure such as loops or branches cannot be detected as such because the DAGs are built up by operator overloading. As a result, the control flow is fixed once the DAG has been built up. Thus, if the code contains statements which cannot be resolved at code generation time, the offline optimization is not applicable. Some flexibility could be obtained if parsing techniques were used rather than operator overloading. If branches could be detected as such, one could extend the scope where the code generator is applicable. Closing the gap to ordinary source code transformation tools would at least mitigate the problem that the outsourced code segment is per se too small in order to pay off. Although branches probably hamper the effectiveness of compiler optimization, some more flexibility would make the code generator approach more viable.

Bibliography

- [1] GCC, the GNU Compiler Collection. *Options That Control Optimization*. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>. Online Documentation, accessed: November 24, 2014.
- [2] Personal conversation with LOTZ, J. on the external function interface of `dco/c++`, RWTH Aachen, Germany, 20. November 2014.
- [3] ANDERSSON, J., ÅKESSON, J., AND DIEHL, M. CasADI: A Symbolic Package for Automatic Differentiation and Optimal Control. In Forth et al. [9], pp. 297–307.
- [4] BISCHOF, C. H., AND BÜCKER, H. M. Computing Derivatives of Computer Programs. In *Modern Methods and Algorithms of Quantum Chemistry* (2000), J. Grotendorst, Ed., vol. 3 of *NIC Series*, John von Neumann Institute for Computing, Jülich, pp. 315–327. Available at <http://webarchiv.fz-juelich.de/nic-series/Volume3/bischof.pdf>.
- [5] BÜCKER, H. M., CORLISS, G. F., HOVLAND, P. D., NAUMANN, U., AND NORRIS, B., Eds. *Automatic Differentiation: Applications, Theory, and Implementations*, vol. 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, 2005.
- [6] CASTAINGS, W., DARTUS, D., HONNORAT, M., LE DIMET, F.-X., LOUKILI, Y., AND MONNIER, J. Automatic Differentiation: A Tool for Variational Data Assimilation and Adjoint Sensitivity Analysis for Flood Modeling. In Bücker et al. [5], pp. 249–262.
- [7] CORBETT, C., MAIER, M., BECKERS, M., NAUMANN, U., GHOBITY, A., AND MITSOS, A. Compiler-Generated Subgradient Code for McCormick Relaxations. Tech. Rep. AIB-2011-25, RWTH Aachen, Germany, November 2011. Available at <http://aib.informatik.rwth-aachen.de/2011/2011-25.pdf>.
- [8] FÖRSTER, M. *Algorithmic Differentiation of Pragma-Defined Parallel Code Regions*. PhD thesis, RWTH Aachen, Germany, 2014. Will appear this year.
- [9] FORTH, S., HOVLAND, P., PHIPPS, E., UTKE, J., AND WALTHER, A., Eds. *Recent Advances in Algorithmic Differentiation*, vol. 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.
- [10] FORTH, S. A., TADJOUDINE, M., PRYCE, J. D., AND REID, J. K. Jacobian Code Generated by Source Transformation and Vertex Elimination Can Be as

- Efficient as Hand-coding. *ACM Transactions on Mathematical Software* 30, 3 (2004), 266–299.
- [11] GIERING, R., KAMINSKI, T., TODLING, R., ERRICO, R., GELARO, R., AND WINSLOW, N. Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model. In Bückner et al. [5], pp. 275–284.
- [12] GRIEWANK, A., AND REESE, S. On the Calculation of Jacobian Matrices by the Markowitz Rule. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, 1991, pp. 126–135.
- [13] GRIEWANK, A., AND WALTHER, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.
- [14] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
- [15] LOTZ, J. Sample code demonstrating how to use the external function interface of dco/c++, RWTH Aachen, Germany. Attached for reference on the CD.
- [16] LOTZ, J., LEPPKES, K., AND NAUMANN, U. dco/c++ – Derivative Code by Overloading in C++. Tech. Rep. AIB-2011-06, RWTH Aachen, Germany, May 2012. Available at <http://aib.informatik.rwth-aachen.de/2011/2011-06.pdf>.
- [17] MARKOWITZ, H. M. The Elimination form of the Inverse and its Application to Linear Programming. *Management Science* 3, 3 (1957), 255–269.
- [18] MEISTER, A. *Numerik linearer Gleichungssysteme. Eine Einführung in moderne Verfahren (Numerical Methods for Linear Systems of Equations. An Introduction to Modern Methods.)*. Vieweg, Wiesbaden, 2008.
- [19] MICHELSEN, M. L. The isothermal flash problem. Part I. Stability. *Fluid Phase Equilibria* 9, 1 (1982), 1–19.
- [20] NAUMANN, U. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University of Dresden, Germany, 1999.
- [21] NAUMANN, U. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Software, Environments, and Tools. SIAM, 2012.
- [22] NAUMANN, U. Combinatorial Problems in Scientific Computing. Lecture slides, RWTH Aachen, Germany, 2013.
- [23] NAUMANN, U., AND LOTZ, J. Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations. Tech. Rep. AIB-2012-10, RWTH Aachen, Germany, June 2012. Available at <http://aib.informatik.rwth-aachen.de/2012/2012-10.pdf>.

- [24] NAUMANN, U., LOTZ, J., LEPPKES, K., AND TOWARA, M. Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations. *ACM Transactions on Mathematical Software*. Submitted for publication.
- [25] ÖZKAYA, E., NEMILI, A., AND GAUGER, N. R. Application of automatic differentiation to an incompressible URANS solver. In Forth et al. [9], pp. 35–45.
- [26] SPEELPENNING, B. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 1980.
- [27] TADJOUDINE, M., BODMAN, F., PRYCE, J. D., AND FORTH, S. A. Improving the Performance of the Vertex Elimination Algorithm for Derivative Calculation. In Bücker et al. [5], pp. 111–120.
- [28] TADJOUDINE, M., FORTH, S. A., PRYCE, J. D., AND REID, J. K. Performance Issues for Vertex Elimination Methods in Computing Jacobians Using Automatic Differentiation. In *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part II* (2002), P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, Eds., vol. 2330 of *Lecture Notes in Computer Science*, Springer, pp. 1077–1086.
- [29] WALTHER, A., GRIEWANK, A., KOWARZ, A., MITEV, H., SCHLENKRICH, S., UTKE, J., AND VOGEL, O. Helmholtz energy function. Available at https://projects.coin-or.org/ADOL-C/browser/stable/2.1/ADOL-C/examples/additional_examples/helm/helm-diff-exam.cpp. Test function of ADOL-C, revision 180.

A

Case Studies

This chapter elaborates on some aspects that have only been adumbrated in Chapter 3. The compilation time, for instance, has not been taken into account because the generation of adjoint source files and their compilation are part of an offline procedure. Section A.1 gives some ideas of how scalable the code generation is in terms of compilation time.

The core statement of the contribution was that the choice of the elimination strategy is not of paramount importance. The multiplication count supposed to be optimized by the elimination strategies takes a back seat to the memory accesses primarily optimized by the compiler. Section A.2 examines the performance of vertex elimination strategies from a theoretical perspective. The focus is in particular on `pathCount`-based strategies.

In Section 3.5 traits of compiler optimization have been discussed. As a conclusion to this section, SSE has been stated to be important for the performance. Section A.3 resumes this discussion with a slightly different focus. Based on the CG method used in Section 3.6, the impact of distinct optimization flags on the performance is analyzed. This should also legitimize why the compiler flags are set to `-O4 -Ofast`.

If not stated otherwise, all measurements in this chapter refer to a system with

compiler:	g++ 4.8.1
CPU:	Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz
L1 cache:	4 × 32 kB instruction cache
	4 × 32 kB data cache
L2 cache:	4 × 256 kB
L3 cache:	8192 kB
RAM:	8 GB.

A.1 Speelpenning's Example

As a classical example in AD, consider Speelpenning's function [26]:

$$f(\mathbf{x}) = \prod_{i=1}^n x_i$$

In [13], this product example is used to motivate AD. An advantage of AD over symbolic differentiation is that common subexpressions can be exploited. The Jacobian matrix

$$\nabla f(\mathbf{x}) = \left(\prod_{i \neq j} x_i \right)_{j=1, \dots, n}$$

can be efficiently accumulated when intermediate results are saved and reused. Here, however, common subexpressions take a back seat. Although vertex elimination deliberately assembles common subexpressions, it is up to the compiler to utilize or ignore the “recommended” accumulation sequence. As seen in the other case studies, compiler optimization takes more optimization parameters into account than merely common subexpressions.

Instead, this example is ideal to examine how scalable the code generation approach is. Figure A.1 depicts the DAG for the following implementation of Speelpenning's product implemented as in Listing A.1.

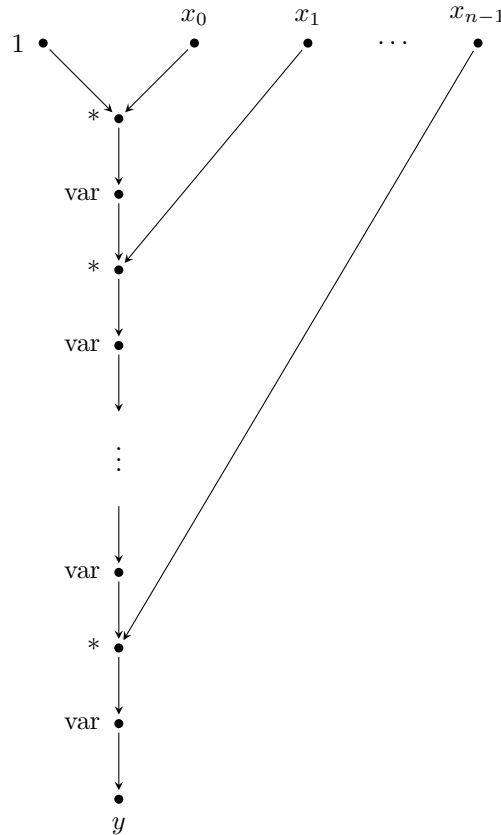


Figure A.1: DAG built up for Speelpenning's example.

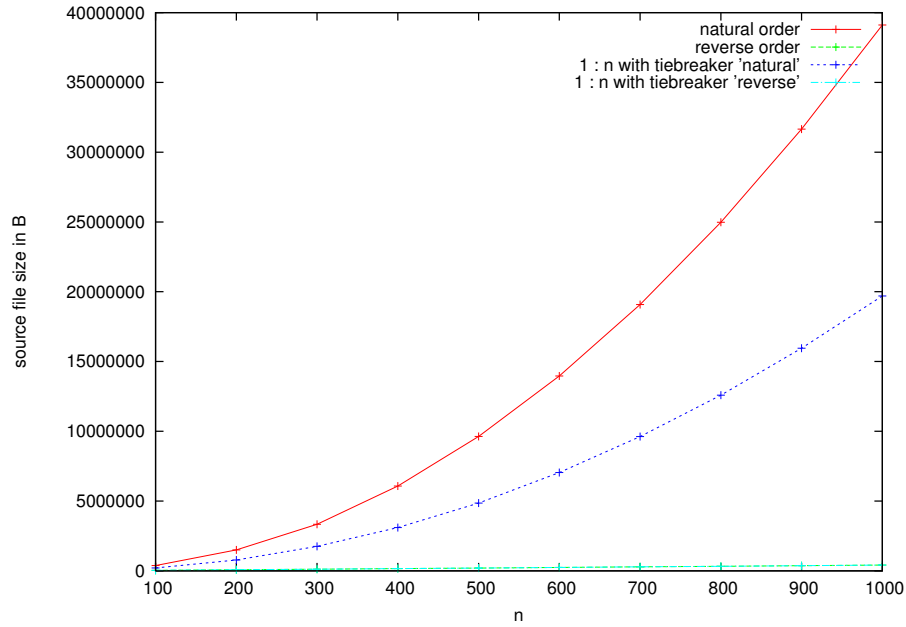


Figure A.2: The size of source files is given as a function of n . Reverse order and $1 : n$ with tiebreaker ‘reverse’ generate source files of identical file size. These source files, however, differ with respect to the order of instructions.

```

1 template <typename T>
2 void f(std::vector<T> &x, std::vector<T> &y) {
3     T res = 1;
4     for(int i=0;i<x.size();i++)
5         res = res * x[i];
6
7     y[0] = res;
8 }

```

Listing A.1: Speelpenning's function.

The variable overwrites respectively assignments to the variable `res` create additional intermediate vertices labeled with `var`. The strategies reverse order and $1 : n$ with tiebreaker ‘reverse’ obtain optimal elimination sequences in terms of a minimal multiplication count whereas elimination according to the strategy natural order yields the worst choice.

Due to the simple structure of the DAG, only four elimination strategies are compared. Figure A.2 verifies that elimination in natural order adds a substantial amount of multiplications. In comparison with the strategy natural order, the strategy $1 : n$ with tiebreaker ‘natural’ induces less edges respectively multiplications of edge labels because $1 : 1$ vertices are eliminated first.

In terms of compiler optimization, optimization choices are limited. As there are direct dependencies between variables rather than independent branches, there are little incentives to reorder instructions. Thus, the primary optimization objective is the reduction of load, store and move instructions while exploiting common subex-

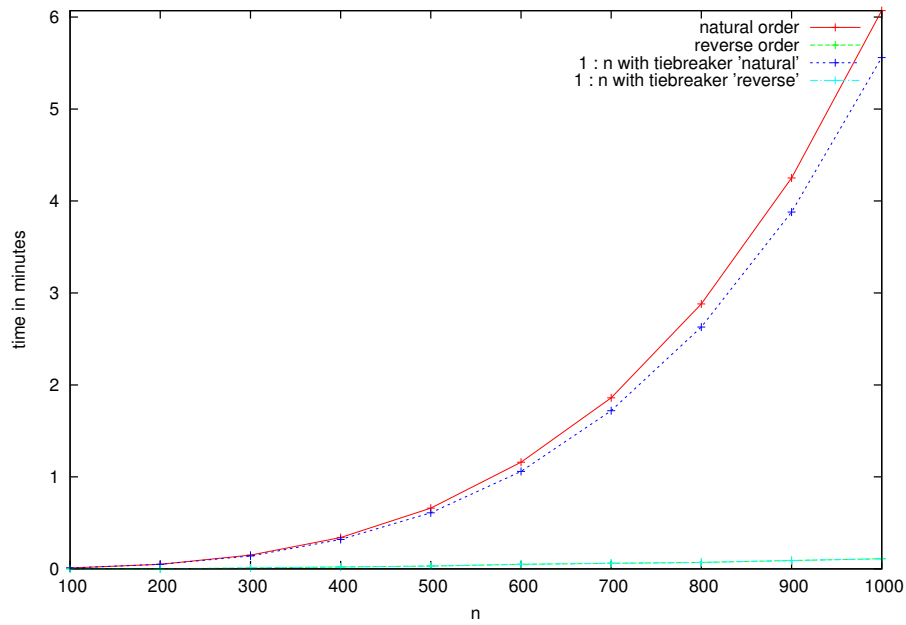


Figure A.3: Compilation time of source code files as a function of n .

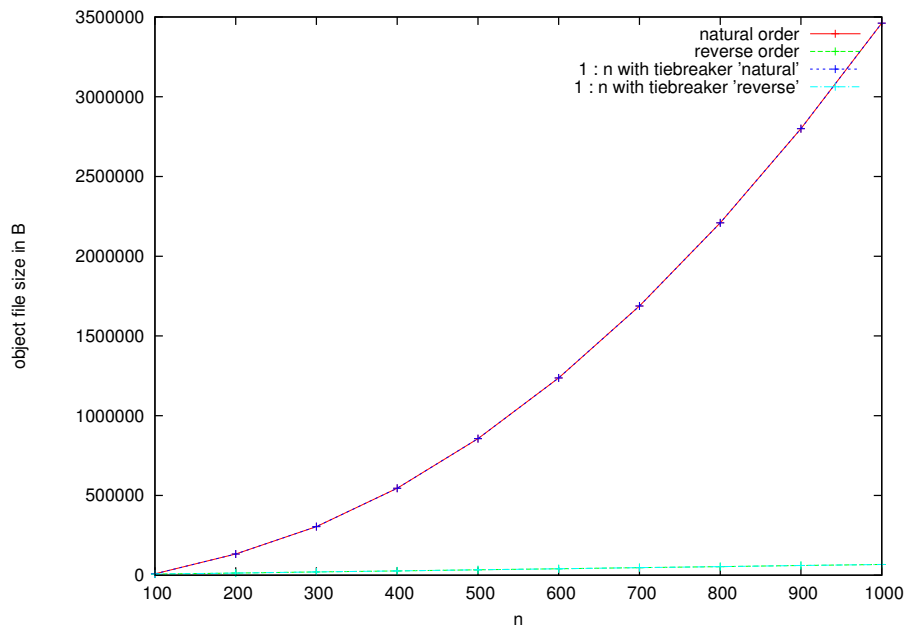


Figure A.4: File size of object files as a function of n .

pressions. The best choice is elimination in reverse order. The prioritized elimination of 1 : 1 vertices has no impact on the compiler's output because multiplications with 1 are removed anyway.

Figure A.3 compares the compilation time of the four source files; the sizes of the resulting object files are shown in Figure A.4. Once again, the compiler equalizes the source files albeit it depends on n to which degree the codes are leveled out. For very small n , compiler optimization even yields the very same object file – no matter

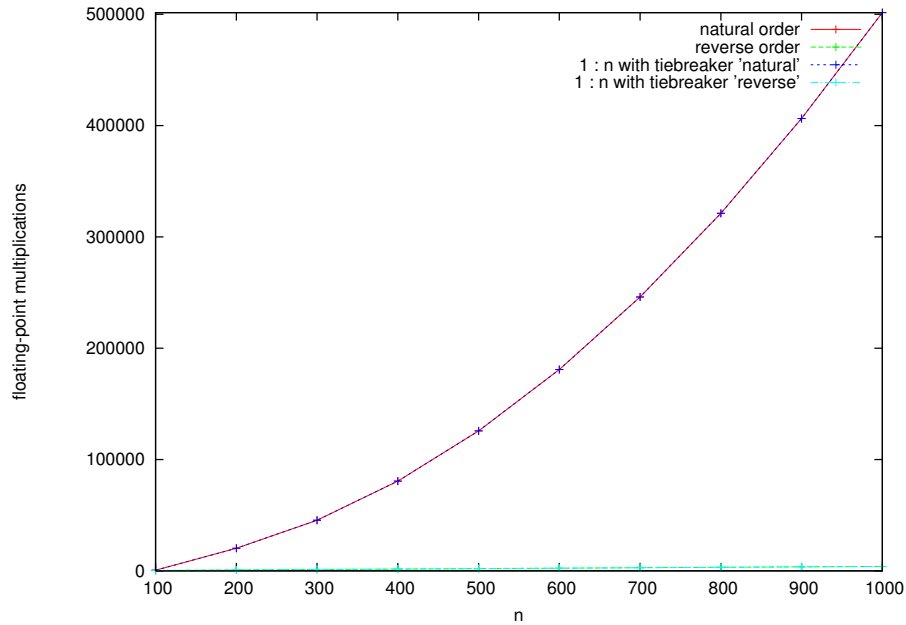


Figure A.5: Number of floating-point multiplications in the object files. Compiler optimization equalizes the codes generated from ‘natural order’ and ‘1 : n with tiebreaker natural’. Similarly, ‘reverse order’ and ‘1 : n with tiebreaker reverse’ are equalized.

which elimination sequence is used. Compiler optimization rearranges the multiplication order. With increasing code size, the extensive search conducted by compiler optimization seemingly is not any longer within the bounds of possibility. Reverse order and 1 : n with tiebreaker ‘reverse’ are still equalized. So are natural order and 1 : n with tiebreaker ‘natural’. The gap between these two groups of strategies widens with increasing n regarding both the compilation time and the object file size. This becomes also apparent in the number of floating-point multiplications shown in Figure A.5.

Admittedly, Speelpenning’s function is an extreme example, but anyway, compiler optimization obviously has a limit. It raises the question of how scalable the code generation through vertex elimination is. In this particular case, elimination in natural order lacks scalability. To remedy this situation, it is possible to split a basic block into smaller segments and treat those separately.

A.2 Reverse Osmosis

Reverse osmosis is a method to desalinate seawater with the aid of semi-permeable membranes. Such membranes cannot be passed by large molecules or ions, but by water molecules. Naturally, water molecules move by diffusion from the side with a lower ion concentration to the side with the higher ion concentration. This movement equalizes the ion concentrations on both sides. Reverse osmosis applies an external pressure to reverse this process. Thereby, water molecules permeate the

strategy and tiebreaker	newly created edges	absorptions	eliminated vertices
natural order	913	73	71
reverse order	99	25	71
static minimum Markowitz			
- random	180	27	71
- natural	758	68	71
- reverse	100	25	71
dynamic minimum Markowitz			
- natural	188	29	71
- reverse	99	25	71
static relative Markowitz			
- random	290	33	71
- natural	487	51	71
- reverse	153	38	71
1 : n			
- natural	762	65	70
- reverse	98	24	70
triangles	109	10	64
average path count			
- random	80	3	32
- natural	100	3	32
- reverse	67	3	32
structural path count	114	11	64

Table A.1: Performance of different strategies on the reverse osmosis example.

membrane, whereas ions are retained on the pressurized side. As a consequence, there is highly concentrated seawater on the pressurized side and water with a lower salt concentration on the other side of the membrane.

In the first place, reverse osmosis is a minimization problem. The efficiency of the desalination process, given by the desalination energy as a function of the water recovery ratio, should be optimized. [7] explains the desalination technique and the optimization problem in greater detail.

The case study, however, uses the code as test case primarily for **pathCount**-based strategies. The DAG has 102 vertices, 71 of which can be eliminated. There is one independent vertex and one dependent vertex, the remainder comprises constants. Constants do not actively contribute to **pathCount**. Although the DAG evidently has several substructures of interest, the presence of a lot of constants may hamper the search for suited starting points.

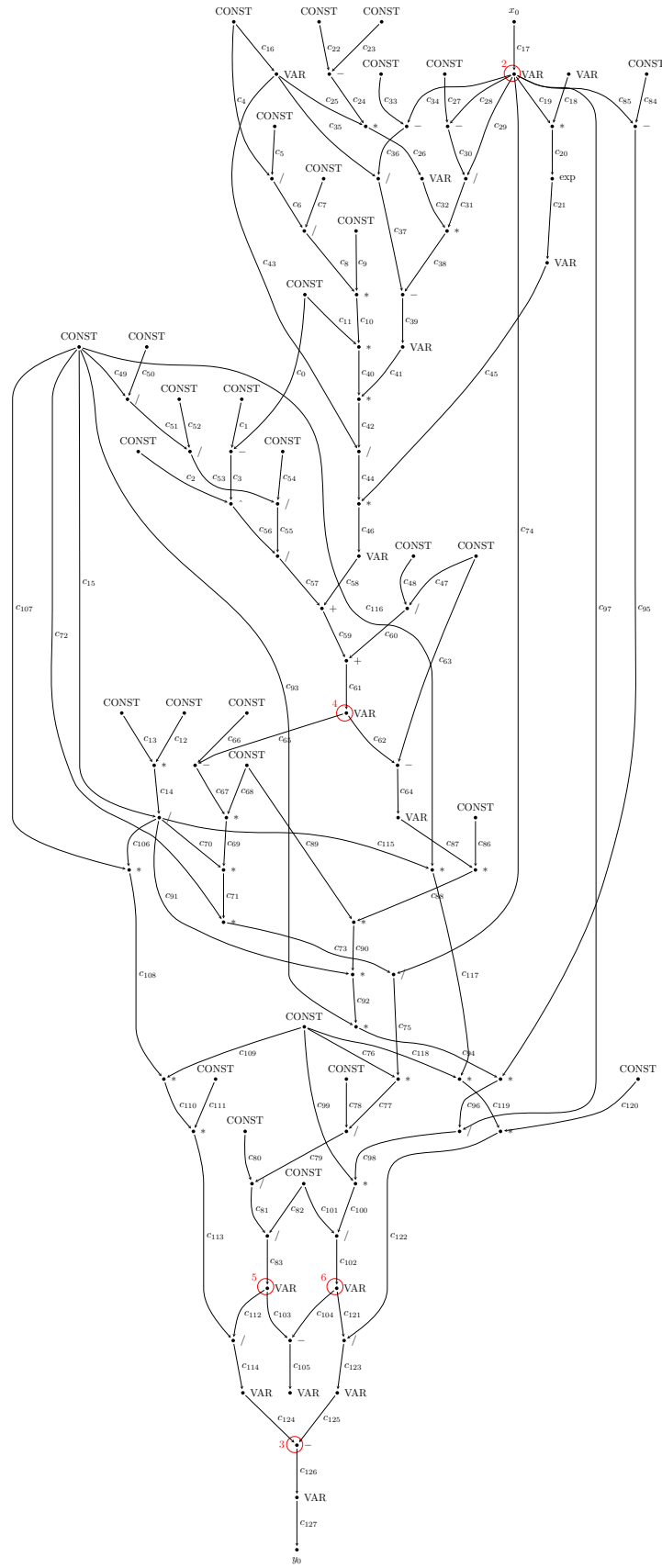


Figure A.6: DAG for the reverse osmosis example. Numbers and circles indicate the order in which vertices are chosen as starting points by structural path count.

Figure A.6 shows the DAG; Table A.1 summarizes how various elimination strategies perform. In general, elimination in reverse order yields better results. Average path count indeed faces problems and falls behind, whereas structural path count performs well.

In relation to the number of vertices actually eliminated, average path count creates a lot of new edges. Constants and vertices whose inputs, maybe indirectly, solely depend on constant vertices constitute large parts of the DAG. These vertices are not targeted during the elimination because their `pathCount` value is zero. Vertices which represent an operation solely dependent on constants, contribute to the overall vertex count when determining the average path count value. Thereby these vertices lower the average path count. As a result, vertices that are (at least in this point in time) better left uneliminated, are eliminated. The compiler de facto mitigates the problem because of constant propagation. If, however, the setup changes and values are not known at compiler time, it may turn out to be a weakness.

In contrast to average path count, structural path count performs well. As a first step, all 1 : 1 vertices are eliminated. Then vertices with high `pathCount` values are searched for. Only a few such vertices are required to serve as starting points for the elimination of triangular substructures. For this DAG, in total six such vertices are found. Those vertices are marked in Figure A.6 in the order in which they are used as starting points. Step 2 determines and eliminates a $\triangle$ -shaped triangle. Step 3 and step 4 eliminate ∇ -shaped triangles. Step 3 is especially effective and affects 55 vertices. The elimination from bottom to top extends the scope to vertices which only depend on constants and not on the independent vertex. As a result, structural path count covers far more vertices than average path count. The DAG contains one dead branch, which misdirects the elimination procedure conducted in step 3. For this reason, the vertices marked in step 5 and step 6 are still part of the DAG. The dead branch comes from a global variable used in the minimization version of reverse osmosis. Therefore, the vertices chosen as starting points in step 5 and step 6 are of no practical use.

The idea of finding suited starting points based on `pathCount` values apparently works out quite well for this example. From a theoretical perspective, structural path count performs well on most examples tested in the course of this thesis. Average path count, by contrast, falls behind if a lot of constants are involved. In practice, these two elimination strategies basically are as good or bad as any other elimination strategy because compiler optimization considers other optimization parameters to be more important than the number of floating-point multiplications.

A.3 Performance and Optimization Flags

This case study experiments with different compiler flags and optimization levels with the intention to justify the choice of `-O4 -Ofast`, which is used throughout this thesis. For this purpose, consider the CG method used in Section 3.6 one more time. There, code segments have been outsourced to benefit from superior adjoint

compiler flags	compilation time	time a1_f()	total runtime
-O0	45 s	2.60 s	3.17 s
-O1	17 min 53 s	0.35 s	0.76 s
-O2	20 min 38 s	0.28 s	0.70 s
-O3	22 min 59 s	0.27 s	0.70 s
-O3 -mno-sse2	20 min 37 s	20.55 s	21.00 s
-O3 -march=native	30 min 32 s	0.32 s	0.73 s
-O4 -Ofast	26 min 23 s	0.21 s	0.65 s
-O4 -Ofast -mno-sse2	20 min 54 s	19.43 s	20.20 s

Table A.2: Impact of compiler flags on compilation time and performance.

code compared to a purely `dco/c++`-based variant. Optimization flags have been set to `-O4 -Ofast` for the compilation of the adjoint code produced by the code generator.

The case study here resumes variant 1 for the outsourced code segment. The elimination strategy used to generate the adjoint code is again structural path count; the dimension of the matrix M_n is equal to $n = 150$. All measurements refer to the same system as in Section 3.6. The focus, however, is now on the impact of different compiler options. Table A.2 summarizes the results obtained for different compiler flags. The compilation time is just for reference and has been timed once. The other timings are based on an average of 10 runs of the executable. The column labeled “time `a1_f()`” lists the fraction of the total runtime spent on the outsourced adjoint code segment.

All of the information in the next paragraphs about control options is based on the official documentation of the `gcc` compiler [1].

The salient point of the measurements is the impact of the activation or the deactivation of SSE2 instructions. The compiler flag `-mno-sse2` deactivates the very basic, scalar SSE2 instructions. SSE2 is a de facto standard hardware feature and is already enabled at optimization level `-O0`. As in variant 1 the computation of matrix-vector product is outsourced, disabling this option degrades the performance.

The performance of a program can potentially be improved by using sophisticated hardware features. The option `-march=native` checks what hardware feature are available and activates those. On the target computer, those feature include `-msse` up to `-msse4.2` and `-mavx`. The latter option can result in further speedup by enabling the AVX encoding of SSE instructions. Compared to the other optimization options, the usage of additional hardware features does not pay off in the CG example.

The change of `-O2` to `-O3` does not have any effect on this specific example. In fact, the compiler’s outputs yield identical object files. Regarding the options `-O3` and `-O4`, `-O4` activates the same flags as `-O3`. For this reason the table does not list any results for the option `-O4`. The optimization option `-Ofast` turns on `-ffast-math`

and disregards standards compliance with IEEE or ISO specifications for math functions.

Already lower optimization levels yield well-optimized object files. The options activated in addition on higher optimization levels do not notably affect the performance. Most of the additional options are probably not applicable to the type of code produced by the code generator because the generated codes exhibit a very simple structure. Enabling the option `-Ofast` yields a small speedup at the expense of disregarding standards compliance.

B

User Documentation

The first part of this chapter describes how to run the examples on the CD. The second part delineates how different adjoint code can be generated and how the generated codes can be integrated into `dco/c++`.

Running the Examples

Quick Guide

As the code generator implements offline optimization, several steps must be executed one after another. Here is a quick guide on the commands to build the executable with the default options.

1. Run `make dag` to build the executable `gen_main`, the code generator.
2. Execute `gen_main` to generate adjoint source files.
3. Run `make gen` to compile the adjoint source files with the default optimization level.
4. Build the executable of the main program. If the main program only evaluates the gradient, run `make test_main`. Otherwise, run `make main`.

Customization

The parameters for the code generation are set in `gen_main.cpp`. Assign a name to the source files to be generated, and specify the vertex elimination strategies to be used. Either input is an argument of the function

```
post(string filename, int strategies, vector<active> &y);
```

Table B.1 lists the naming of the elimination strategies. If linked with bitwise ORs, several source files can be generated at once. Running `make dag` creates, if necessary, the directory `generatedCode/` and then builds the executable `gen_main`. The execution of `gen_main` creates both header and source files in accordance to the elimination strategies specified. Source and header files are produced in the directory `generatedCode/` and extended with a number identifying the elimination strategy applied by the code generator.

Example B.0.1. Running first `make dag` and then executing `gen_main` with the specification

```
post(adjoint_f, REVERSE | STATIC_RELATIVE_MARKOWITZ, y);
```

in `gen_main.cpp` creates the files

- (a) `adjoint_f_1.cpp` and `adjoint_f_1.hpp` implementing `REVERSE` and
- (b) `adjoint_f_4.cpp` and `adjoint_f_4.hpp` for `STATIC_RELATIVE_MARKOWITZ`.

Running `make gen` compiles all source files located in `generatedCode/`. As indicated in the previous chapters, it can take a substantial amount of time until the compiler has compiled all source files.

In order to use a certain adjoint code, include the corresponding header file in the main program. Moreover, the Makefile definition `ADJOINT :=` specifies which adjoint code shall be linked and must be adapted accordingly. All examples contain a readme file listing further parameters that can be changed like for instance the dimension of the input vectors. Running `make main` or `make test_main` builds the executable.

Overview of Makefile Targets

Depending on the example, only a subset of the commands is available.

- **make dag:** creates the directory `generatedCode/` and builds the executable named `gen_main` for the code generation.
- **make gen:** compiles all files in `generatedCode/`.
- **make test_main:** builds an executable that abstracts from the original problem and solely evaluates the Jacobian.
- **make main:** builds the “default” main program.
- **make dco_only_main:** for comparison reasons, some examples exhibit a variant which only uses `dco/c++`.
- **make sd:** builds steepest descent solver – only for minimization problems such as the reverse osmosis example.
- **make clean:** removes all executables and object files except for the adjoint object files in `generatedCode/`.

- **make superclean**: extends **make clean** by removing `generatedCode/`, which entails the removal of all adjoint object, source and header files produced by the code generator.

strategy	ID	description
NATURAL	0	order in which vertices are created is equal to the elimination order
REVERSE	1	reversed natural order, corresponds to reverse mode
STATIC_MINIMUM_MARKOWITZ	2	Markowitz degree computed statically, vertex with lowest Markowitz degree is eliminated first
DYNAMIC_MINIMUM_MARKOWITZ	3	Markowitz degree recomputed after each vertex elimination
STATIC_RELATIVE_MARKOWITZ	4	static minimum Markowitz degree enhanced with a dependency degree to penalize vertices whose Markowitz degree is likely to grow
THRESHOLD_MARKOWITZ_6	5	computes Markowitz degree statically and only eliminates vertices with a Markowitz degree ≤ 6 .
ONE_TO_MANY_NATURAL	6	1 : n with tiebreaker ‘natural’
ONE_TO_MANY_REVERSE	7	1 : n with tiebreaker ‘reverse’
TRIANGLES	8	eliminates 1 : 1 vertices, then 1 : n vertices in natural order and 2 : 1 in reverse order
STRUCT_PATH_COUNT	10	uses <code>pathCount</code> values to identify central vertices, then eliminates triangular substructures
AVG_PATH_COUNT	11	eliminates all vertices whose <code>pathCount</code> value is below the average <code>pathCount</code> value.
ALL		convenience macro to generate source files for all available strategies
PATHCOUNT		convenience macro comprising strategies based on <code>pathCount</code> values
MARKOWITZ		convenience macro covering Markowitz-based strategies

Table B.1: Vertex elimination strategies.

Usage as AD Tool

The employment of the code generator in conjunction with `dco/c++` follows a two-step process. First, the code segment intended to be outsourced must be called with an `active` type to generate to corresponding adjoint source code. Second, the adjoint source code must be registered via the external function interface of `dco/c++`.

At the code generation stage, the function signature of the code segment which should be outsourced must match the following format.

```
template <typename T>
void f(const std::vector<T> &x, std::vector<T> &y) {
    ... // y = f(x)
}
```

For simplicity, consider the function to be given as a separate file called `f.hpp`.

The actual code generation requires a `gen_main`, a separate program solely used for the code generation. In `gen_main`, it is specified what strategies are applied, what functions are to be differentiated, and what the dimension of the vectors is. A sample is shown in Listing B.1.

```
1 #include <vector>
2 #include "dag_driver.hpp"
3 #include "active.hpp"
4 #include "dag.hpp"
5 #include "f.hpp"
6
7 extern dag our_dag;
8
9 int main(int argc, char **argv) {
10     const int n = ...;
11     const int m = ...;
12     std::vector<active> x(n);
13     std::vector<active> y(m);
14
15     //no init
16
17     setup(x);
18     f<active>(x,y);
19     post("filename", ALL, y);
20
21     return 0;
22 }
```

Listing B.1: Sample `gen_main.cpp` for the code generation. The macro `ALL` generates code for each of the elimination strategies.

The function calls `setup()` and `post()` enclose the section of the code for which adjoint code is generated. If `setup()` and `post()` enclose several function calls or other computations, these code fragments are treated as a unit. In practice, it is most certainly better to not glue computations together in order to avoid problems with large file sizes. All entries of the vector `x` are registered as independent variables;

analogously, all entries of the vector y become dependent variables. Figure B.1 illustrates the interaction of the program components involved in the code generation. An overview of the available elimination strategies is given in Table B.1.

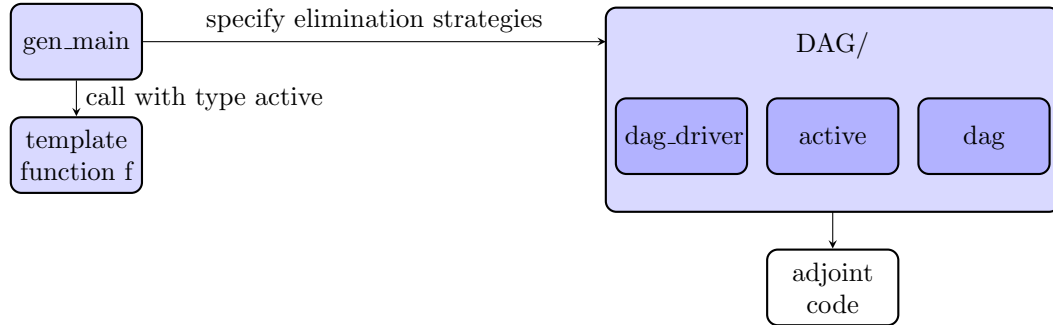


Figure B.1: Interaction of components needed for the code generation.

The execution of `gen_main` generates adjoint source files, which are intended to be compiled using a high optimization level.

Next, the adjoint code from the code generator shall be used within `dco/c++`. A minimalistic, artificial example, which demonstrates how to integrate the generated code into `dco/c++`, is also attached on the CD under `genericExample/`. In contrast to the code given here, the dimensions of the outsourced subfunction and the global function do not coincide.

Listing B.2 demonstrates how the adjoint source codes are accessed within `dco/c++`. The notation used for the comments is the same as in Section 3.6. Referencing [2, 24], `create_ext_fcn_data()` (line 39) registers the adjoint code from the code generator. In the reverse run, this function is called when the interpretation of the tape reaches the gap in the tape. The function calls `register_input()` (line 42) and `register_output()` (line 45) enclose the outsourced code segment and copy the values. Furthermore, these functions allow filling the gap in the reverse run properly: On the one hand, the correct seed can be fetched (line 16). On the other hand, the result computed in `a1_f()` is written back in such a way that the correct global adjoints are incremented (line 21). Since `a1_f()` recomputes intermediate results, values required to do so are saved in the checkpoint (line 43) and fetched in the reverse run (line 12).

```

1 #include <vector>
2 #include "f.hpp"
3 #include "generatedCode/adjoint_f_10.hpp" //adjoint source file
4 #include "dco.hpp"
5 using namespace std;
6 using namespace dco;
7
8 const int n = ...;
9 const int m = ...;
10
11 void adjoint_f(gais<double>::external_function_data *data) {
12     vector<double> px; data->read_from_checkpoint(px); //fetch  $\hat{x}_{(1)}$ 
13     vector<double> a1_x(n), a1_y(m);
  
```

```

14
15     for(int i=0;i<m;i++)
16         a1_y[i] = data->get_output_adjoint();    //fetch  $\hat{y}_{(1)}$ 
17
18     a1_f(px, a1_x, a1_y);    //returns  $\hat{x}_{(1)} \equiv a1\_x$ 
19
20     for(int i=0;i<n;i++)
21         data->increment_input_adjoint(a1_x[i]); //  $x_{(1)} = x_{(1)} + \hat{x}_{(1)}$ 
22 }
23
24 int main(int argc, char** argv) {
25     vector<gals<double>::type> x(n), y(m);
26     //init x
27     ...
28
29     //tape
30     gals<double>::global_tape = gals<double>::tape::create();
31     gals<double>::global_tape->register_variable(x);
32
33     //implement function  $y = F(x)$ 
34     //compute something
35     ...
36
37     //prepare for outsourced code segment
38     auto *T = get_tape(x[0]);
39     auto *data = T->create_ext_fcn_data<gals<double>::
40         external_function_data>(adjoint_f);
41     vector<double> px(n); //  $\hat{x}$ 
42     vector<double> py(m); //  $\hat{y}$ 
43     data->register_input(x,px);
44     data->write_to_checkpoint(px);
45     f<double>(px,py); //  $\hat{y} = f(\hat{x})$ 
46     data->register_output(py,y);
47
48     //compute something else
49     //...
50
51     //forward run done
52
53     //reverse
54     gals<double>::set(y[...], 1.0, -1); //seed  $y_{(1)}$ 
55     gals<double>::global_tape->interpret_adjoint();
56
57     //harvest derivatives
58     ...
59     return 0;
60 }

```

Listing B.2: Sample program showing how the generated adjoint files are used by means of the external function interface of `dco/c++`; the function calls and the structure of the program follow [15].

C

Developer Documentation

The next paragraphs provide a brief overview of the developer documentation. A complete documentation can be found on the attached CD.

Keeping Track of Changes in the DAG

Two global data structures, namely the vector **nodes** for the vertices and the vector **edges**, are in use to represent the DAG. The size of **nodes** is constant throughout the code generation. Rather than physically eliminating a vertex, a flag **alive** indicates whether or not a vertex is still present. The vector **edges** maintains edge objects. The size of this vector can only increase in the course of an elimination procedure, but never decrease. In contrast to the vertices, there is no explicit flag to signal the status of an edge. Instead, an edge is still present if it is accessible via a vertex. For this purpose, every vertex maintains one vector for incoming edges and one for outgoing edges. These vectors do not store the actual edge, but the *position* of the edge object in **edges**. The position is equal to the ID of an edge. Analogously, the position in **nodes** corresponds to the ID of a vertex. Figure C.2 gives an (incomplete) overview of the most important components using UML-inspired notation.

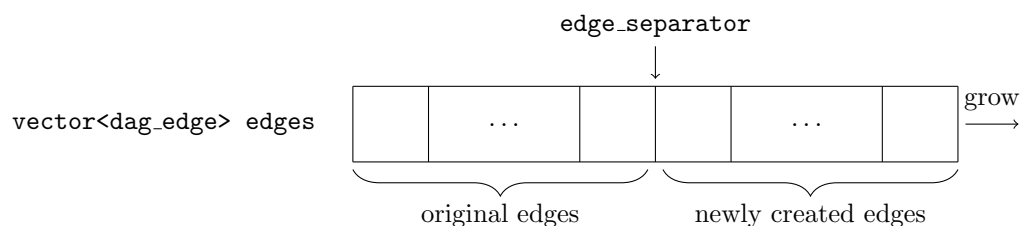


Figure C.1: Data structure for the edges.

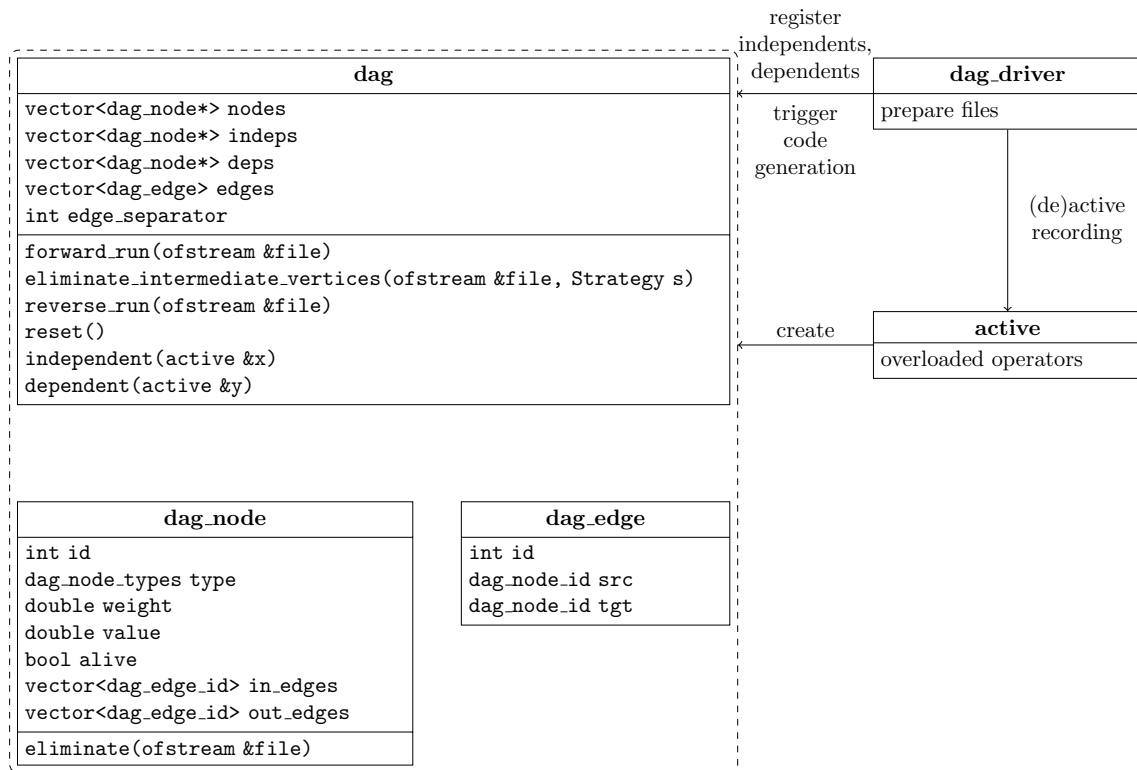


Figure C.2: Components (abridged) of the code generator and their interaction.

The reason why the DAG is basically maintained as a global data structure is due to generation of several source files in a row. If several source files are supposed to be generated at a time, the DAG must be reset to its initial state to apply distinct elimination strategies. With regard to the vertices, the task of restoring the original DAG is easy to accomplish; the flag **alive** is reset for all vertices in **nodes**. When it comes to restoring the edges, a variable **edge_separator** saves the position of the last edge belonging to the original DAG. All edges beyond this position have been created in the elimination procedure. Thus, any edge beyond this position is deleted if the original DAG is restored. Figure C.1 illustrates this. At last, the lists of ingoing and outgoing edges owned by a vertex can be reconstructed from **edges** up to the position marked by **edge_separator**. This approach is viable because only the existence of an edge, but not its actual label is stored.

Regarding Figure C.2, some attributes of **dag_node** may not be self-explanatory.

- **id**: As the ID of a vertex is equal to its position in the vector, this variable is actually superfluous. It is however kept for convenience reason as it allows for generating variable names quickly.
- **type**: the type of operation like for instance $+$, $*$, $\sin$, var or const . The type var represents vertices induced by the assignment operator. The type const represents constants, which are unique in the sense that the actual value must be stored.
- **weight**: cost assessment such as the Markowitz degree. Some strategies, for example structural path count, do not make use of this variable.

- **value**: This attribute is only used to assign the correct value to variables representing constants in the source file and ignored otherwise.

An elimination procedure is a sequence of three function calls with the strategy being the only difference. Recall how the generated source files look like, which was exemplarily explained in Section 3.3.

1. **forward_run()**: writes the implementation of the forward run to the file.
2. **eliminate_intermediate_vertices()**: determines the elimination sequence according to some elimination strategy. For each vertex about to be eliminated, the function **eliminate()** is called, which records any changes of the edge labels in the source file. If new edges are created in the process, the data structure **edges** is extended. Eliminated vertices are marked as such via **alive**.
3. **reverse_run()**: The remainder of the DAG is traversed from the dependents up to the independents. Edges accessed in the process give the variable IDs required for accumulating the adjoints.

Source Code and Additional Examples

The content of the attached CD is structured as depicted in Figure C.3. Most examples either have been presented in Appendix A or have been part of the Contribution. The additional examples comprise:

- **heat/**. The integration of outsourced code into **dco/c++** is demonstrated with the one-dimensional heat equation.

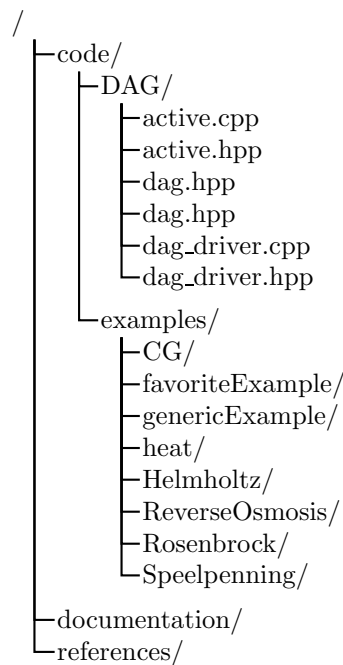


Figure C.3: Content of the attached CD.

- **Rosenbrock/**. The method of steepest descent is used to approximate the global minimum of Rosenbrock's function. For this purpose, the Jacobian matrix is computed once per iteration. The code generator is used to differentiate Rosenbrock's function. This example does not involve `dco/c++`.
- **genericExample/**. This tiny, artificial example tackles the integration of outsourced code segments into `dco/c++`.