Universidade Estadual de Campinas
Instituto de Matemática Estatística e Computação Científica
Departamento de Matemática Aplicada

Diferenciação Automática de Matrizes Hessianas

Robert Mansel Gower*
Mestrado em Matemática Aplicada - Campinas - SP

Orientador: Profa. Dra. Margarida P. Mello

∗ Este trabalho teve apoio financeiro da FAPESP processo 2009/04785-7.

Universidade Estadual de Campinas
Instituto de Matemática Estatística e Computação Científica
Departamento de Matemática Aplicada

Automatic Differentiation of Hessian Matrices

Robert Mansel Gower*
Master in Applied Mathematics - Campinas - SP.

Supervisor: Profa. Dra. Margarida P. Mello

∗ This project was funded by FAPESP 2009/04785-7.

Diferenciação Automática de Matrizes Hessianas

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defindida por **Robert Mansel Gower** e aprovada pela comissão julgadora.

Campinas, 10 de Maio de 2011

Margarida P. Mello

Profa. Margarida Pinheiro Mello
Orientadora

Banca Examinadora

1. Prof. Dr. Ernesto Julián Goldberg Birgin (IME – USP)
2. Prof. Dr. Aurélio Ribeiro Leite de Oliveira (IMECC – UNICAMP)
3. Profa. Dra. Margarida Pinheiro Mello (IMECC – UNICAMP)

Dissertação apresentada ao Instituto de Matemática, Estatística e Computação Científica, Unicamp, como requisito parcial para obtenção do Título de MESTRE em Matemática Aplicada.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Gower, Robert Mansel
G747d Diferenciação automática de matrizes hessianas/Robert Mansel Gower-- Campinas, [S.P. : s.n.], 2011.

Orientador : Margarida Pinheiro Mello
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1.Matriz hessiana. 2.Diferenciação automática. 3.Matrizes esparsas. 4.Teoria dos grafos. I. Mello, Margarida Pinheiro. II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. III. Título.

Título em inglês: Automatic differentiation of hessian matrices

Palavras-chave em inglês (Keywords): 1.Hessian matrix. 2.Automatic differentiation. 3.Sparse matrices. 4.Graph theory.

Área de concentração: Análise Numérica

Titulação: Mestre em Matemática Aplicada

Banca examinadora: Profa. Dra. Margarida Pinheiro Mello (IMECC - UNICAMP)
Prof. Dr. Ernesto JuliánGoldberg Birgin (IME- USP)
Prof. Dr. Aurélio Ribeiro Leite de Oliveira (IMECC - UNICAMP)

Data da defesa: 15/04/2011

Programa de Pós-Graduação: Mestrado em Matemática Aplicada

Dissertação de Mestrado defendida em 15 de abril de 2011 e aprovada

Pela Banca Examinadora composta pelos Profs. Drs.

Margarida P. Mello

Prof.(a). Dr(a). MARGARIDA PINHEIRO MELLO

Aurelio R L de Oliveira

Prof.(a). Dr(a). AURELIO RIBEIRO LEITE DE OLIVEIRA

Prof.(a). Dr(a). ERNESTO JULIÁN GOLDBERG BIRGIN

O formato atual desta dissertação foi aprovado pela orientadora e pela CPG do Instituto de Matemática, Estatística e Computação Científica.

Profa. Dra Margarida Pinheiro Mello
Orientadora

Prof. Dr. Luiz Koodi Hotta
CPG do IMECC

Long Live Nerv

Strike me... and I shall smight thee,
Hesitate... and I shall obliterate,
Differentiate... and I shall intergrate.
Always poised to undo the evil doing that has been done!

Since you have left me,
my heart has been `SEGMENTATION FAULT`.
Since you said goodbye,
all I can say is `abort, cancel or retry`.

Remember not to neglect the importance
of properly `allocating` ones sentiments,
for debugging is a pain.

–Nervinator.

Preface

Coffee, approximately 2160 cups, and 4320 hours of work have somehow materialized this dissertation. The journey into the unknown was frustrating at times, for I didn't know what to do. But an almost irrational persistence and the guiding hand of my supervisor Margarida P. Mello pulled me through. Clearly science is an activity to be done in groups (of at least two people), otherwise we would all be slowly trudging down the dark tunnels of deductive reasoning, where it is too dark to see the false premise at the burrow's mouth.

I thank my loving parents, who have always given me multidimensional support, encouraging me to do whatever my heart desired, may it be dance, music or mathematics. Luckily, it was mathematics. I thank my father for showing me a way of looking at the world around us, and questioning what we see. Via example, my mother taught me to be audacious and to challenge what is accepted by others.

Last but not least, I thank a rather peculiar assortment of individuals: Nerv. Though it is time for us to disband, Nerv will always be at the epicentre of our hearts and the current location of its members. May we reach the summits of our ambitions, and remember: never conform, never compromise.

Robert Mansel Gower.

Campinas, March 2011.

Resumo

Dentro do contexto de programação não linear, vários algoritmos resumem-se à aplicação do método de Netwon aos sistemas constituídos pelas condições de primeira ordem de Lagrange. Nesta classe de métodos é necessário calcular a matriz hessiana. Nosso foco é o cálculo exato, dentro da precisão da máquina, de matrizes hessianas usando diferenciação automática. Para esse fim, exploramos o cálculo da matriz hessiana sob dois pontos de vista. O primeiro é um modelo de grafo que foca nas simetrias que ocorrem no processo do cálculo da hessiana. Este ângulo propicia a intuição de como deve ser calculada a hessiana e leva ao desenvolvimento de um novo método de modo reverso para o cálculo de matrizes hessianas denominado `edge_pushing`. O segundo ponto de vista é uma representação puramente algébrica que reduz o cálculo da hessiana à avaliação de uma expressão. Esta expressão pode ser usada para demonstrar algoritmos já existentes e projetar novos. Para ilustrar, deduzimos dois novos algoritmos, `edge_pushing` e um novo algoritmo de modo direto, e uma série de outros métodos conhecidos [1], [20, p.157] e [9].

Apresentamos estudos teóricos e empíricos sobre o algoritmo `edge_pushing`. Analisamos sua complexidade temporal e de uso de memória. Implementamos o algoritmo como um *driver* do pacote ADOL-C [19] e efetuamos testes computacionais, comparando sua performance com à de dois outros *drivers* em dezesseis problemas da coleção CUTE [5]. Os resultados indicam que o novo algoritmo é muito promissor.

Pequenas modificações em `edge_pushing` produzem um novo algoritmo, `edge_pushing_sp`, para o cálculo da esparsidade de matrizes hessianas, um passo necessário de uma classe de métodos que calculam a matriz hessiana usando colorações de grafos, [14, 19, 30]. Estudos de complexidade e testes numéricos são realizados comparando o novo método contra um outro recentemente desenvolvido [30] e os testes favorecem o novo algoritmo `edge_pushing_sp`.

No capítulo final, motivado pela disponibidade crescente de computadores com multiprocesadores, investigamos o processamento em paralelo do cálculo de matrizes hessianas. Examinamos o cálculo em paralelo de matrizes hessianas de funções parcialmente separáveis. Apresentamos uma abordagem desenvolvida para o cômputo em paralelo que pode ser usado em conjunto com qualquer método de cálculo de hessiana e outra estratégia específica para métodos de modo reverso. Testes são executados em um computador com memória compartilhada usando a interface de programação de aplicativo OpenMP.

Keywords: hessian matrix, automatic differentiation, sparse matrices, graph theory.

Abstract

In the context of nonlinear programming, many algorithms boil down to the application of Newton's method to the system constituted by the first order Lagrangian conditions. The calculation of Hessian matrices is necessary in this class of solvers. Our focus is on the exact calculation, within machine precision, of Hessian matrices through automatic differentiation. To this end, we detail the calculations of the Hessian matrix under two points of view. The first is an intuitive graph model that focuses on what symmetries occur throughout the Hessian calculation. This provides insight on how one should calculate the Hessian matrix, and we use this enlightened perspective to deduce a new reverse Hessian algorithm called `edge_pushing`. The second viewpoint is a purely algebraic representation of the Hessian calculation via a closed formula. This formula can be used to demonstrate existing algorithms and design new ones. In order to illustrate, we deduce two new algorithms, `edge_pushing` and a new forward algorithm, and a series of other known Hessian methods [1], [20, p.157] and [9].

We present theoretical and empirical studies of the `edge_pushing` algorithm, establishing memory and temporal bounds, and comparing the performance of its computer implementation against that of two algorithms available as drivers of the software ADOL-C [14, 19, 30] on sixteen functions from the CUTE collection [5]. Test results indicate that the new algorithm is very promising.

As a by-product of the `edge_pushing` algorithm, we obtain an efficient algorithm, `edge_pushing_sp`, for automatically obtaining the sparsity pattern of Hessian matrices, a necessary step in a class of methods used for computing Hessian matrices via graph coloring, [14, 19, 30]. Complexity bounds are developed and numerical tests are carried out comparing the new sparsity detection algorithm against a recently developed method [30] and the results favor the new `edge_pushing_sp` algorithm.

In the final chapter, motivated by the increasing commercial availability of multiprocessors, we investigate the implementation of parallel versions of the `edge_pushing` algorithm. We address the concurrent calculation of Hessian matrices of partially separable functions. This includes a general approach to be used in conjunction with any Hessian software, and a strategy specific to reverse Hessian methods. Tests are carried out on a shared memory computer using the OpenMP paradigm.

Keywords: hessian matrix, automatic differentiation, sparse matrices, graph theory.

Contents

List of Algorithms

Chapter 1

Introdução

Ao examinar uma função na vizinhança de um ponto, podemos descartar várias características complicadas da função e aproximá-la por funções mais simples. A mais conhecida destas aproximações é a expansão de primeira ordem de Taylor. Seja f uma função real que toma vetores do $\mathbb{R}^n$ como entrada e cujas derivadas de primeira ordem são contínuas, ou seja, $f \in C(\mathbb{R}^n, \mathbb{R})$. Denominamos gradiente o vetor coluna das derivadas de primeira ordem de $f(x)$,

$$\nabla f(x) = \begin{pmatrix} \dfrac{\partial f}{\partial x_1}(x) \\ \vdots \\ \dfrac{\partial f}{\partial x_n}(x) \end{pmatrix}.$$

Usando o gradiente, podemos construir uma função afim que, em uma vizinhança próxima de $x \in \mathbb{R}^n$, serve como uma aproximação razoável da nossa função:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T (\Delta x). \tag{1.1}$$

Pode-se melhorar esta aproximação adicionando informação de segunda ordem: a matriz hessiana.

$$f''(x) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}.$$

É um fato bem conhecido que, se as derivadas de segunda-ordem f forem contínuas, isto é, $f \in C^2(\mathbb{R}^n, \mathbb{R})$, então a sua matriz hessiana será simétrica. A aproximação de segunda ordem é uma função quadrática:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T (\Delta x) + (\Delta x)^T f''(x)(\Delta x). \tag{1.2}$$

Para ilustrar, seja $f(x,y) = \cos(x)\cos(y)$. Logo, a sua aproximação de segunda ordem nos arredores da origem:

$$\begin{aligned} h(x,y) &= f(0) + \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\ &= 1 - x^2 - y^2. \end{aligned} \tag{1.3}$$

A vantagem de se usar essa aproximação é que funções quadráticas são muito mais simples de se entender. Repare que (2.3) possui um máximo local na origem, e, consequentemente, a função $f(x,y)$ também.

Tais aproximações (2.1) e (2.2) possuem várias aplicações. Por exemplo, a caracterização de pontos de equilíbrio e análise de sensibilidade [6, 11, 25]. Outro contexto em que matrizes hessianas são utilizadas é no método de Netwon para resolver sistemas Lagrangianas não lineares. Um sistema não linear é um conjunto de equações não lineares

$$\begin{aligned} F_1(x) &= 0 \\ F_2(x) &= 0 \\ &\vdots \\ F_m(x) &= 0, \end{aligned} \tag{1.4}$$

onde $F_i : \mathbb{R}^n \to \mathbb{R}$. Usando uma notação vetorial podemos escrever (2.4) como $F(x) = 0$, onde $F(x) = (F_1(x), F_2(x), \ldots, F_m(x))^T$. Suponha que $F \in C^2(\mathbb{R}^n, \mathbb{R}^m)$. Se trocamos F em (2.4) pela sua aproximação linear na vizinhaça de um ponto x^0, obtemos

$$F(x^0) + F'(x^0)(x - x^0) = 0. \tag{1.5}$$

Onde F' é uma matriz tal que a i-ésima linha é o gradiente transposto de F_i. Equação (2.5) pode ser reescrita como

$$F'(x^0)x = F'(x^0)x^0 - F(x^0).$$

Suponha que $F'(x^0)$ seja uma matriz não singular e seja x^1 a solução do sistema linear acima. Podemos repetir estes passos (aproximar e depois resolver o sistema linear) e assim construir uma sequência (x^i) tal que x^{i+1} é a solução de

$$F'(x^i)x^{i+1} = F'(x^i)x^i - F(x^i).$$

Se (x^i) converge, então

$$\|F(x^i)\| = \|F'(x^i)(x^{i+1} - x^i)\| \leq \|F'(x^i)\| \, \|(x^{i+1} - x^i)\| \to 0,$$

portanto, $(F(x^i))$ converge a zero e a sequência (x^i) converge para a solução de (2.4). Este método é conhecido como o método de Newton.

Dentro do contexto de otimização não linear, um ponto de solução deve satisfazer um determinado sistema não linear. Um problema básico de otimização não linear é

$$\begin{aligned} &\min f(x) \\ &\text{sujeito a } h(x) = 0, \end{aligned} \tag{1.6}$$

onde $f \in C^2(\mathbb{R}^n, \mathbb{R})$ e $h \in C^2(\mathbb{R}^n, \mathbb{R}^m)$. A condição de otimalidade de primeira ordem para um determinado x é que exista $\lambda \in \mathbb{R}^m$ tal que

$$\nabla f(x) - \sum_{i=1}^{m} \lambda_i \nabla h_i(x) = 0. \tag{1.7}$$

Para resolver (2.7), o método de Netwon é comumente usado. Substituindo as funções em (2.7) pelas suas aproximações lineares nos arredores do ponto (x^0, λ^0), obtemos

$$\nabla f(x^0) - \sum_{i=1}^{m} \lambda_i^0 \nabla h_i(x^0) + \left(f''(x^0) - \sum_i \lambda_i^0 h_i''(x^0) \right) (x - x^0) - h'(x^0)(\lambda - \lambda^0) = 0. \tag{1.8}$$

Este sistema linear contêm duas matrizes hessianas – $f''(x^0)$ e $h''(x^0)$. O método dos pontos interiores, ubíquo em software de otimização não linear [10], usa variantes do método de Newton. Enquanto os pacotes de otimização LOQO [28] exigem que o usuário forneça a matriz hessiana, IPOPT [29] e KNITRO [7] são mais flexíveis, mas também utilizam de uma forma ou outra informação da matriz hessiana. Portanto, a necessidade de se calcular a matriz hessiana é impulsionada pela popularidade crescente dos métodos de otimização que se aproveitam de informação de segunda ordem.

Nem sempre é necessário calcular as matrizes hessianas $f''(x^0)$ e $h''(x^0)$ inteiras para resolver o sistema (2.8). Dependendo da forma escolhida para resolver (2.8), possivelmente apenas produtos hessiana-vetor são necessários, como no método do gradiente conjugado. Torna-se necessária uma análise de caso a caso para decidir se vale a pena calcular a matriz hessiana inteira ou apenas um conjunto de produtos hessiana-vetores. Em 1992, um método reverso eficiente de differenciação automática (AD) foi desenvolvido para calcular produtos hessiana-vetor [9]. Reverso, porque os cálculos são realizados na ordem inversa em relação à avaliação da função subjacente. O software de otimização KNITRO [7] oferece a opção de calcular somente produtos hessiana-vetor.

Existe uma extensão deste método hessiana-vetor reverso usado para calcular a matriz hessiana inteira [20, p.155], porém, talvez o método mais comun encontrado na literatura, que parece ter surgido no trabalho de Jackson e McCormick [23], seja o algoritimo hessiana direto. A principal desvantagem deste algoritmo direto é a sua complexidade temporal que é n^2 vezes a do cálculo da função subjacente, onde n é a dimensão do domínio da função.

Uma abordagem verdadeiramente bem-sucedida, que calcula matrizes hessianas esparsas, combina diferenciação automática de produtos hessian-vetor e coloração de grafos [14, 30]. De modo sucinto, estes métodos calculam uma versão comprimida da matriz hessiana usando coloração de grafos e o algoritmo AD hessiana-vetor, e subsequentemente "descomprime", obtendo assim a matriz hessiana original.

Os cinco primeiros capítulos desta dissertação foram condensados em artigo científico, cujo título é "A new framework for the computation of Hessians". O mesmo foi submetido e posteriormente aceito para publicação no *Optimization Methods and Software*.

Chapter 2

Hessian Matrix: Why Calculate?

When examining a function in a neighborhood of a point, one can discard many complicated aspects of the function and accurately approximate it with simpler functions. The best known of these approximations is the first-order Taylor expansion. Let f be a real valued function whose input is an n-dimensional vector, with first-order partial derivatives, thus $f \in C(\mathbb{R}^n, \mathbb{R})$. The column vector of the first-order partial derivatives of $f(x)$ is called the gradient

$$\nabla f(x) = \begin{pmatrix} \dfrac{\partial f}{\partial x_1}(x) \\ \vdots \\ \dfrac{\partial f}{\partial x_n}(x) \end{pmatrix}.$$

Using the gradient, one can obtain an affine function that serves as a reasonable approximation in a close neighborhood of the (fixed) point $x \in \mathbb{R}^n$:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T(\Delta x). \tag{2.1}$$

One can improve this approximation by adding second-order information, the Hessian matrix.

$$f''(x) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}.$$

It is well known that if f possesses continuous second order partial derivatives, in other words $f \in C^2(\mathbb{R}^n, \mathbb{R})$, then its Hessian matrix is symmetric. The second-order approximation is a quadratic function:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T(\Delta x) + (\Delta x)^T f''(x)(\Delta x). \tag{2.2}$$

To illustrate, let $f(x,y) = \cos(x)\cos(y)$ and its second order approximation around the origin:

$$\begin{aligned} h(x,y) &= f(0) + \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\ &= 1 - x^2 - y^2. \end{aligned} \tag{2.3}$$

The advantage of using such an approximation is that quadratic functions are far simpler to understand. Note that it is immediate that (2.3) has a local maximum around the origin, and thus so does the surface it approximates.

Such approximations (2.1) and (2.2) have many applications, a few examples are characterizing equilibrium points and sensitivity analysis [6, 11, 25]. Another context in which Hessian matrices are used is Newton's method for solving nonlinear Lagrangian systems. A nonlinear system is a set of nonlinear equations

$$\begin{aligned} F_1(x) &= 0 \\ F_2(x) &= 0 \\ &\vdots \\ F_m(x) &= 0, \end{aligned} \tag{2.4}$$

where $F_i : \mathbb{R}^n \to \mathbb{R}$. Using vectorial notation (2.4) becomes $F(x) = 0$, where $F(x) = (F_1(x), F_2(x), \ldots, F_m(x))^T$. Suppose $F \in C^2(\mathbb{R}^n, \mathbb{R}^m)$. If we replace F in (2.4) with its linear approximation in the vicinity of a point x^0, then the system becomes

$$F(x^0) + F'(x^0)(x - x^0) = 0. \tag{2.5}$$

Here F' is a matrix where the i-th row is the transposed gradient of F_i. Equation (2.5) may be rewritten as

$$F'(x^0)x = F'(x^0)x^0 - F(x^0).$$

Suppose that $F'(x^0)$ is an invertible matrix, and let x^1 be the solution of the above linear system. One can repeat these steps (approximate and solve a linear system) to build a sequence (x^i) such that each x^{i+1} is the solution of

$$F'(x^i)x^{i+1} = F'(x^i)x^i - F(x^i).$$

If (x^i) converges then

$$\|F(x^i)\| = \|F'(x^i)(x^{i+1} - x^i)\| \leq \|F'(x^i)\| \, \|(x^{i+1} - x^i)\| \to 0,$$

thus $(F(x^i))$ converges to zero and the sequence (x^i) tends to a solution of (2.4). This method is known as Newton's method.

Within the context of nonlinear optimization, a solution point must satisfy a certain nonlinear system. A basic nonlinear optimization problem is

$$\begin{aligned} &\min f(x) \\ &\text{subject to } h(x) = 0, \end{aligned} \tag{2.6}$$

where $f \in C^2(\mathbb{R}^n, \mathbb{R})$ and $h \in C^2(\mathbb{R}^n, \mathbb{R}^m)$. The first-order optimality conditions for x is that there exist a $\lambda \in \mathbb{R}^m$ such that

$$\nabla f(x) - \sum_{i=1}^{m} \lambda_i \nabla h_i(x) = 0. \tag{2.7}$$

Newton's method is commonly employed to solve (2.7). Replacing the functions in (2.7) with their linear approximation around the point (x^0, λ^0), we obtain

$$\nabla f(x^0) - \sum_{i=1}^{m} \lambda_i^0 \nabla h_i(x^0) + \left(f''(x^0) - \sum_i \lambda_i^0 h_i''(x^0) \right) (x - x^0) - h'(x^0)(\lambda - \lambda^0) = 0. \tag{2.8}$$

This linear system contains two Hessian matrices $f''(x^0)$ and $h''(x^0)$. Interior-point methods, ubiquitous in nonlinear solvers [10], use variants of Newton's method. While the nonlinear optimization package LOQO [28] requires that the user supply the Hessian, IPOPT [29] and KNITRO [7] are more flexible, but also use Hessian information of some kind or other. Thus the need to efficiently calculate Hessian matrices is driven by the rising popularity of optimization methods that take advantage of second-order information.

It is not always necessary to obtain Hessian matrices $f''(x^0)$ and $h''(x^0)$ to solve system (2.8). Depending on how one chooses to solve (2.8), one might only require Hessian-vector products, as is the case with the conjugate gradient method. It becomes very much a case-by-case analysis to decide whether it is worth calculating the entire Hessian matrix as opposed to a set of Hessian-vector products. In 1992, a time efficient reverse Automatic Differentiation (AD) method was developed for calculating Hessian-vector products [9]. Reverse, for the calculations are carried out in the reverse order in which would evaluate the underlying function. The optimization solver KNITRO [7] offers the option of calculating only Hessian-vector products.

There also exists an extension of this Hessian-vector method for the entire Hessian matrix [20, p.155], though, perhaps the most common method found in the literature, apparently appearing first in Jackson and McCormick's work [23], is the Forward Hessian algorithm. The major drawback of this forward algorithm is its heavy time penalty, presenting a complexity of n^2 times the function evaluation's complexity, where n is the dimension of the function's domain.

A truly successful approach for calculating sparse Hessian matrices combines the automatic differentiation of Hessian-vector products with graph coloring [14, 30]. Simply put, these methods calculate a compressed version of the Hessian matrix using graph colouring and the Hessian-vector AD procedure, to then subsequently "decompress" it and obtain the original.

Chapter 3

Introduction to Automatic Differentiation

3.1 What is and is not Automatic Differentiation

There are three well-known methods in use for calculating derivatives: Finite Differences, Symbolic Differentiation and Automatic Differentiation. We will start with what is not AD (Automatic Differentiation). AD calculates derivatives within floating point precision, as opposed to approximations. This is already different from Finite Differences where one would use the limit definition to approximate partial derivatives, e.g, given a function $f : \mathbb{R}^n \to \mathbb{R}$, one can approximate the gradient of f at $x^0 \in \mathbb{R}^n$ with n function evaluation such as:

$$\frac{\partial f}{\partial x_i}(x^0) \approx \frac{f(x^0 + he_i) - f(x^0)}{h}.$$

Symbolic differentiation is also exact, but there is a difference between the input and output as compared to automatic differentiation. Symbolic differentiation's input is a function f and its output is another function, the derivative f'. In addition, this f' output of symbolic differentiation has to be something the user can "touch" and manipulate and looks like the derivative we would see in a calculus course. Take the function $f(x_1, x_2, x_3, x_4) = x_1 \sin(x_2) x_3 \cos(x_4)$, for example. The first derivative in this case is the gradient ∇f. The symbolic differentiation output would be something that resembles:

$$\nabla f(x) = \begin{pmatrix} \sin(x_2) x_3 \cos(x_4) \\ x_1 \cos(x_2) x_3 \cos(x_4) \\ x_1 \sin(x_2) \cos(x_4) \\ -x_1 \sin(x_2) x_3 \sin(x_4) \end{pmatrix}. \tag{3.1}$$

Typically, in AD one does not strive for a palpable semblance of f' and is concerned solely with fast evaluations of f' on a given point. Thus many AD routine's input is a function f and point x, and its output is $f'(x)$. Though there exists a form of AD called source code

transformation which returns the source code of a subroutine that calculates f' on any given point, but this code is unintelligible at a glance unlike the example above.

If all that one wants is to evaluate numerical values of ∇f, say in the context of an optimization routine, using (3.1) can be very inefficient. Note how the partial derivatives shares common subexpressions. For instance, $x_1 \sin(x_2)$ appears in two partial derivatives, hence many calculations would be repeated using the above expression. In contrast to (3.1), the Black-Box Gradient in Algorithm 3.1 does not repeat unnecessary calculations.

Algorithm 3.1: Black-Box Gradient $\nabla f(x)$

Input: (x_1, x_2, x_3, x_4)

$$
\begin{aligned}
c &= \cos(x_4) \\
s &= \sin(x_2) \\
a &= x_3 c \\
b &= x_1 s \\
D_3 f &= bc \\
D_1 f &= as \\
a &= x_1 a \\
b &= x_3 b \\
D_4 f &= -a \sin(x_4) \\
D_2 f &= b \cos(x_2)
\end{aligned}
\quad ;
$$

Output: $D_1 f, D_2 f, D_3 f, D_4, f$

The Black-Box Algorithm is much like a AD routine, for its output is the gradient evaluated on a point, and gives you no representation of the function f' itself. To the user this might indeed appear to be a bit of a black-box. The advantage is that while evaluating the gradient using our previous symbolic formula we would calculate: ten products, four $\sin(x)$ and four $\cos(x)$ evaluations, the Black-Box Gradient algorithm performs: eight products, two $\sin(x)$ and two $\cos(x)$ evaluations.

In AD, great savings are made by not repeating common threads of calculation. So much so, that it will be shown that evaluating the gradient has the same complexity as evaluating the underlying function.

3.2 The Function and Notation

We will restrict ourselves to functions that can be expressed as a composition of a fixed set of "known" functions. We refer to this set of known functions as the set of elemental functions, for they are the basic building blocks of which our functions are composed. A large variety of functions can be described as a composition of but a few elementary functions ,e.g.,

$$f(x_1, x_2, x_3) = \cos(x_1)\frac{3x_2}{x_1} + 3x_2 \exp(x_3). \tag{3.2}$$

In this example $f(x)$ is a composition of the functions $\cos(x)$, $\exp(x)$, division, multiplication and summation. If we know the derivative of each elemental function, using the chain rule we can derive by hand the functions under consideration. Automatic differentiation works on this principle, hence AD packages are accompanied by a library of elemental functions and their corresponding derivatives already coded.

To be able to talk about how to differentiate a function, we have to define precisely how it will be represented, or, in other words, what is the input for an automatic differentiation algorithm.

We define a *function program* as a procedure that a compiler knows how to evaluate. A typical example in C++ or FORTRAN would look like Algorithm 3.2. calculations.

Algorithm 3.2: A Function Program

Input: (x_1, x_2, x_3)

$$f(x_1, x_2, x_3) = \cos(x_1) * \left(\frac{3x_2}{x_1}\right) + 3 * x_2 * \exp(x_3)$$

Output: $f(x_1, x_2, x_3)$

Although Algorithm 3.2 is user friendly, the order in which the terms are evaluated is compiler dependent, in contrast to the list of statements detailed in the Alternative Evaluation Procedure in Algorithm 3.3, where the order is fixed and the appearance is less appealing to the user. Furthermore, if one were to differentiate (3.2), for each application of the chain-rule there is one line and internal variable v_i of Algorithm 3.3.

Algorithm 3.3: An Alternative Evaluation Procedure for (3.2)

Input: (x_1, x_2, x_3)

$v_1 = \cos(x_1)$

$v_2 = 3 * x_2$

$v_3 = v_2/x_1$

$v_4 = v_2 * v_3$

$v_5 = \exp(x_3)$

$v_6 = v_2 * v_5$

$v_7 = v_6 + v_4$

Output: v_7

Such a list of internal variables as in Algorithm 3.3 is soon to become commonplace in this dissertation. So that the numbering scheme of the variables reflect their order of evaluation, and other reasons yet to be clarified, we found it convenient to apply, throughout this dissertation, a shift of $-n$ to the indices of all matrices and vectors. We already have $x \in \mathbb{R}^n$, which, according to this convention, has components x_{1-n}, x_{2-n}, ..., x_0. Similarly, the rows/columns of the Hessian f'' are numbered $1-n$ through 0. Other vectors and matrices

will be gradually introduced, as the need arises for expressing and deducing mathematical properties enjoyed by the data.

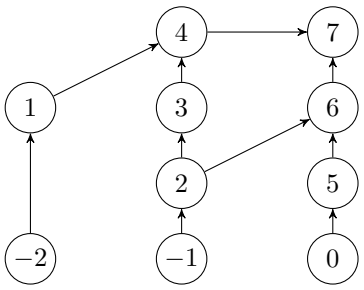
A graphic and more intuitive way of representing the list evaluation procedure in Algorithm 3.3 is as a directed acyclic graph called the computational graph.

A graph $G = (V, E)$ is comprised of a set V of nodes, or vertices, and a set of pares of nodes E called edges or arcs. If an edge (i, j) is an ordered pair, then we say the edge is directed, while unordered pairs $\{i, j\}$ are referred to as undirected edges. The pictorial depiction of a directed edge has an arrowhead, as in Figure 3.1, while undirected edges have not. Let $S(i) = \{j \mid \exists (i, j) \in E\}$ be the set of successors of node i and $P(i) = \{j \mid \exists (j, i) \in E\}$ be the set of its predecessors. An example of a directed graph is in Figure 3.1.

Formally, a computational graph is a pair $\mathcal{CG} = (G, \varphi)$ where $G = (V \cup Z, E)$ is a directed graph and $\varphi = (\phi_1, \ldots, \phi_\ell)$ a list of elemental functions. $Z = \{1 - n, \ldots, 0\}$ is the set of nodes with indegree zero, and $V = \{1, \ldots, \ell\}$ is the set of intermediate nodes. The node $\ell \in V$ is called the dependent node, and is special in that it is the only node with outdegree zero. Each node is associated to a function as follows: node $i - n \in Z$ represents the ith independent variable, so $v_{i-n} = x_{i-n}$, and to nodes $i \in V$ we associate $v_i = \phi_i\left(v_{P(i)}\right)$, a elemental function of its predecessors. The nodes are numbered so that if $j \in P(i)$ then $j < i$. For each node i in V, there is a path from some node with indegree zero to node i. $\mathcal{CG}$ can be seen as a device to orderly evaluate several composite functions, assigning values to all intermediate nodes, given a set of values for the independent variables, in one forward sweep of the graph, as follows:

$$\begin{aligned} v_{i-n} &= x_{i-n}, && \text{for } i = 1, \ldots, n \\ v_i &= \phi_i(v_{P(i)}), && \text{for } i = 1, \ldots, |V|. \end{aligned}$$

From this point of view, each node $i \in V$ is ultimately associated to a function of the independent variables, say $v_i = u_i(x)$. As with many other formal definitions, it is common to relax the formality when working with the defined objects. In this case, this means identifying the computational graph with its "graph" part, failing to make explicit mention of the functions. We will adhere to this lax use if no ambiguity derives therefrom. Figure 3.1 gives an example of such a graph.



$$\begin{aligned} v_1 &= \cos(x_{-2}) \\ v_2 &= 3 * x_{-1} \\ v_3 &= v_2 / x_{-2} \\ v_4 &= v_2 * v_3 \\ v_5 &= \exp(x_0) \\ v_6 &= v_2 * v_5 \\ v_7 &= v_6 + v_4 \end{aligned}$$

Figure 3.1: A Computational Graph and an evaluation procedure of the function $f(x_{-2}, x_{-1}, x_0) = \cos(x_{-2})\,(3x_{-1}/x_{-2}) + 3x_{-1}\exp(x_0)$.

A commonly used notation is the precedence symbol $\prec$, where $j \prec i$ denotes that v_j is part of the input of ϕ_i, thus $j \prec i \Leftrightarrow j \in P(i)$. Both forms of notation will be used, using the predecessor $P(i)$ and successor $S(i)$ sets in graph contexts, but the two notations are interchangeable.

Algorithm 3.4: An Evaluation Procedure of a Function $f(x) = y$.

Input: $x \in \mathbb{R}^n$.
for $i = 1 - n, \ldots, 0$ **do**
    $v_i = x_i$
end
for $i = 1, \ldots, \ell$ **do**
    $v_i = \phi_i(v_j)_{j \prec i}$
end
Output: v_ℓ

Using the precedence notation $\prec$ the evaluation of a function is described as in Algorithm 3.4. The first loop copies the current values of the independent variables $x_{1-n}, ..., x_0$ into the variables $v_{1-n}, ..., v_0$. The actual function evaluation takes place in the next loop with ℓ elemental function evaluations ϕ_i, $i = 1, \ldots, \ell$, which are stored in ℓ internal intermediate variables v_i, $i = 1 \ldots \ell$. The precedence notation hides the number of input variables, in that $\phi_i(v_j)_{j \prec i}$ denotes the evaluation of ϕ_i on all v_j variables of which ϕ_i depends, e.g., if ϕ_i is a binary function, and $P(i) = \{j, k\}$, then $\phi_i(v_j)_{j \prec i} = \phi_i(v_j, v_k)$. The last step copies the intermediate variable v_ℓ into the output variable y.

Any function can be unrolled into a (potentially long) sequential evaluation of one elemental function and one floating point value per line as in Algorithm 3.4. This sort of representation is called the factored form by Jackson and McCormick [23], and Wengert List in [2] and [32]. We will refer to it as the list of intermediate functions. Note that, if the time to carry-out a look-up and calculate an elemental function ϕ_i is bounded by a constant, then the complexity of evaluating f with ℓ intermediate functions is bounded by a constant multiple of ℓ. Of course the user is not required to supply the function in such a format. On the contrary, there will be an interface that transforms the user supplied function program into a list of intermediate variables. This automatic conversion can be carried out by operator overloading. We will expand on how this is done in the Chapter 5.

3.3 Calculating The Gradient Using The Computational Graph

With automatic differentiation one can obtain the gradient of a function at a small fixed multiple of the cost of evaluating the underlying function, a complexity that does not depend explicity on the dimension of the domain. Attempts at calculating the partial derivatives

$\partial f/\partial x_i$, for $i = 1, \ldots, n$, as n separate functions gives the impression that the cost of calculating $\nabla f(x)$ depends on n, when in fact the calculation of each $\partial f/\partial x_i$ can share many threads with the calculation of other partial derivatives. This is the key to realizing that the cost of a gradient evaluation can be of the same order as that of a function evaluation, although there may be overheads.

Here we present a known graph interpretation of an efficient gradient routine, followed in the next two sections by an algebraic representation. This format of presentation, graph then algebra, is repeated with the Hessian.

In Calculus textbooks the computation of partial derivatives via the chain rule is often explained with the help of a diagram, see, for instance, [26, p. 940], where (intermediate) variables are replicated as need, so that the diagram corresponding to the evaluation of the function is a tree, see Figure 3.2. The leaves of the trees are associated with the independent variables, possibly replicated, and the top node with the function.

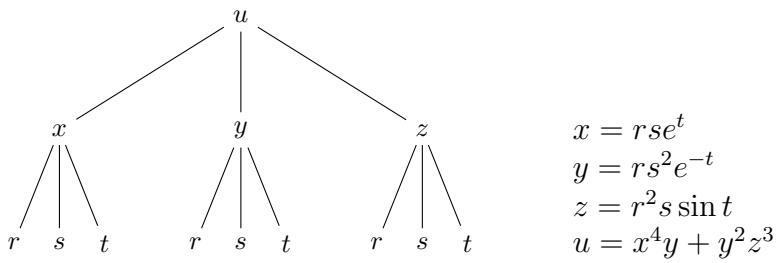


Figure 3.2: Tree diagram in example 5 of [26, p. 940].

Thus the partial derivative of the function with respect to one of the variables is obtained by moving up the tree from each leaf that is a replica of this variable to the top, multiplying the appropriate partial derivatives. For instance, there are three copies of t in Figure 3.2, which provides three paths from a 't' to the top and the partial derivative in t is:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial u}{\partial z}\frac{\partial z}{\partial t}.$$

Notice that the partial derivative associated with an arc in the diagram depends only on the arc's end points, and we may think of it as a weight associated with the arc. Then each path from a leaf to the top node also has a weight, which is the product of the weights of the arcs in the path. The desired partial derivative is the sum of the weights of these paths, one from each replica of the variable to the top node.

Now a computational graph may be seen as obtained from the tree diagram by merging the nodes that were replicas of each other. Usually this means that the graph is no longer a tree. Nevertheless, the same computations may be carried out, taking into account that there may be several paths from a zero indegree node (the leaves of the tree diagram) to node ℓ (previously the top node). Therefore the first order partial derivative of f with respect to one of the variables, say x_{i-n}, will simply be the sum of the weights of all the paths from

node $i-n$ to node ℓ in G:

$$\frac{\partial f}{\partial x_{i-n}} = \sum_{p|\text{path from } i-n \text{ to } \ell} \left(\prod_{(i,j)\in p} \frac{\partial \phi_j}{\partial v_i} \right) = \sum_{p|\text{path from } i-n \text{ to } \ell} \left(\prod_{(i,j)\in p} c_j^i \right), \tag{3.3}$$

where c_j^i is the weight associated with arc $(i,j) \in E$.

Different algorithms for calculating the gradient can be perceived as different ways of accumulating these edge weights. As an example, let $G = (V \cup Z, E)$ be a computational graph of a function $f : \mathbb{R}^n \to \mathbb{R}$. Let $|V| = \ell$. Assume that the functions $\phi_i \in C^2$, for $i = 1, \ldots, \ell$. Then the first order partial derivatives of f may be computed by a backward sweep of the computational graph, as prescribed in Algorithm 3.5, [20]. If the only concern (or possibility) is the numerical evaluation of functions, then the backward sweep must be preceded by a forward sweep, where the values associated with the intermediate nodes, corresponding to a given set of values for the independent values, are computed. Thus, in the backward sweep, the partial derivatives $\partial \phi_j / \partial v_i$ will be evaluated at the points computed in the forward sweep. But the algorithm is self-sufficient if one is doing symbolic computations.

Algorithm 3.5: Reverse Gradient Calculation

Input: $G = (V \cup Z, E)$, $|V| = \ell$, $\{\phi_i \mid i = 1, \ldots, \ell\}$
Initialization $\bar{v}_\ell = 1$, $\bar{v}_i = 0$, for $i = 1-n, \ldots, \ell - 1$
for $i = \ell, \ldots, 1$ **do**

$$\bar{v}_j = \sum_{j \in P(i)} \bar{v}_i \frac{\partial \phi_i}{\partial v_j}(v_{P(i)})$$

end
Output: $\{\bar{v}_{i-n} \mid 1 \le i \le n\}$

Algorithm 3.5 is referred to as reverse accumulation and as the Non-incremental Adjoint Recursion by Griewank and Walther [20, p. 41]. From the description of the computation using tree diagrams and chain rule, it is straightforward to conclude that the algorithm is the implementation of the tree diagram method adapted to the computational graph. In particular, notice that, after node i is swept, the variable $\bar{v}_i$ contains the sum of the weights of the paths from i to ℓ. Therefore, at the end of the algorithm $\bar{v}_{i-n} = \partial f / \partial x_{i-n}$, for $i = 1, \ldots, n$. The graph interpretation gives us an intuitive view of the chain rule, but we leave the more formal correctness proof for the next section, where we use an algebraic representation.

3.4 The Function as State Transformations

Writing the function as a loop is very convenient computer-wise, but quite awkward for demonstration purposes. In order to derive a one line representation for a function let

$$\mathrm{v}_i := (v_{1-n}, \ldots, v_i, 0, \ldots, 0) \in \mathbb{R}^{n+\ell}, \quad \text{for } i = 0, \ldots, \ell. \tag{3.4}$$

To each elemental function ϕ_i a state transformation

$$\begin{array}{rccc} \Phi_i: & \mathbb{R}^{\ell+n} & \to & \mathbb{R}^{\ell+n} \\ & (y_{1-n},\ldots,y_i,\ldots,y_\ell) & \mapsto & (y_{1-n},\ldots,\phi_i(y_j)_{j\prec i},\ldots,y_\ell) \end{array}$$

is associated. In other words, Φ_i mimics the identity function in all coordinates except the ith coordinate, where it sets y_i to $\phi_i(y_j)_{j\prec i}$. By sequentially applying the state transformations we can build the vectors defined in (3.4) as such:

$$\begin{array}{rcl} \mathrm{v}_1 & = & \Phi_1(\mathrm{v}_0), \\ \mathrm{v}_i & = & \Phi_i(\mathrm{v}_{i-1}), \quad i = 1\ldots,\ell. \end{array}$$

With these definitions we can write $f(x)$ in a single equation:

$$f(x) = e_\ell^T \left(\Phi_\ell \circ \Phi_{\ell-1} \circ \cdots \circ \Phi_1\right)(P^T x), \tag{3.5}$$

where e_ℓ is the $(\ell+n)$th canonical vector and $P = [I_{n\times n}, 0, \ldots, 0]$. This representation of $f(x)$ as a series of state transforms was introduced by Griewank and Walther [20].

3.5 Calculating the Gradient Using State Transformations

Differentiating and recursively using the chain rule on (3.5) we arrive at:

$$\nabla f^T = e_\ell^T \Phi_\ell' \Phi_{\ell-1}' \cdots \Phi_1' P^T. \tag{3.6}$$

For simplicity's sake, the argument of each function is omitted in (3.6), but it should be noted that Φ_k' is evaluated at $(\Phi_{k-1} \circ \Phi_{k-2} \circ \cdots \circ \Phi_1)(P^T x)$, for $k = 1, \ldots, \ell$.

Each intermediate variable may also be written as a composition of state transformations and seen as a function of the independent variables:

$$v_j(x) = e_j^T \left(\Phi_j \circ \Phi_{j-1} \circ \cdots \circ \Phi_1\right)(P^T x), \tag{3.7}$$

thus the gradient of each intermediate variable may also be written as:

$$\nabla v_i^T = e_i^T \Phi_i' \Phi_{i-1}' \cdots \Phi_1' P^T. \tag{3.8}$$

Although one can present each gradient differentiation algorithm in this chapter and prove correctness without the aid of (3.6), it provides a broader perspective, in the sense that it may be used to design and demonstrate several algorithms. With this in mind, multiplying the Jacobians Φ_i' in a left-to-right fashion yields the Block Reverse Gradient Algorithm 3.6, while multiplying from the right-to-left yields the Block Forward Gradient Algorithm 3.7. A previous forward sweep is carried out in each algorithm to calculate each Φ_i and store the resulting intermediate variables, so in turn we can evaluate Φ_i'.

Algorithm 3.6: Block Reverse Gradient Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\bar{v} \leftarrow e_{\ell+n}$
(*forward sweep to calculate each* Φ_i)
for $i = \ell, \ldots, 1$ **do**
  $\bar{v}^T \leftarrow \bar{v}^T \Phi_i'$
end
Output: $\nabla f(x)^T = \bar{v}^T P^T$

Algorithm 3.7: Block Forward Gradient Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\dot{V} \leftarrow P^T \in \mathbb{R}^{(n+\ell)\times n}$
(*forward sweep to calculate each* Φ_i)
for $i = 1, \ldots, \ell$ **do**
  $\dot{V} \leftarrow \Phi_i' \dot{V}$
end
Output: $\nabla f(x)^T = e_{\ell+n}^T \dot{V}$

We will refer to algorithms that perform a sequential reverse sweep of the intermediate variables from v_ℓ to v_1 as reverse mode algorithms, and algorithms that perform forward sweeps as forward mode algorithms. To perform the calculations in the Block Reverse Gradient Algorithm, an auxiliary vector $\bar{v} \in \mathbb{R}^{\ell+n}$ is used, called the adjoint vector. We shall refer to $\bar{v}_i$ as the i-th adjoint value.

The translation to a componentwise computation of the vector-matrix products of Algorithm 3.6 is very much simplified by the special structure of the Jacobian of the state transformation Φ_i. This follows from the fact that the function in component j of Φ_i is given by

$$[\Phi_i]_j(y) = \begin{cases} y_j, & \text{if } j \neq i, \\ \phi_i(y_j)_{j \prec i}, & \text{if } j = i. \end{cases} \tag{3.9}$$

Since row j of the Jacobian Φ_i' is the transposed gradient of $[\Phi_i]_j$, we arrive at the following block structure for Φ_i':

$$\Phi_i' = \left[\begin{array}{c|c|c} & & \\ I & 0 & 0 \\ & & \\ \hline & & \\ (c^i)^T & 0 & 0 \\ & & \\ \hline & & \\ 0 & 0 & I \\ & & \end{array}\right] \begin{array}{l} 1-n \\ \vdots \\ i-1 \\ \text{row } i, \\ i+1 \\ \vdots \\ \ell \end{array} \tag{3.10}$$

where

$$c_j^i = \frac{\partial \phi_i}{\partial v_j}, \quad \text{for } j = 1-n, \ldots, i-1. \tag{3.11}$$

Thus $(c^i)^T$ is basically the transposed gradient of ϕ_i padded with the convenient number of zeros at the appropriate places. In particular, it has at most as many nonzeros as the number of predecessors of node i, and the post-multiplication $\overline{\mathrm{v}}^T\Phi_i'$ affects the components of $\overline{\mathrm{v}}$ associated with the predecessors of node i and zeroes component i. In other words, denoting component i of $\overline{\mathrm{v}}$ by $\bar{v}_i$, the block assignment in Algorithm 3.6 is equivalent to

$$\bar{v}_j \leftarrow \begin{cases} \bar{v}_j + \bar{v}_i \dfrac{\partial \phi_i}{\partial v_j}, & \text{if } j \prec i, \\ 0, & \text{if } j = i, \\ \bar{v}_j, & \text{otherwise.} \end{cases}$$

Now this assignment is done as the node i is swept, and, therefore, in subsequent iterations component i of $\overline{\mathrm{v}}$ will not be accessed, since the loop visits nodes in decreasing index order. Hence setting component i to zero has no effect on the following iterations. Eliminating this superfluous reduction, we arrive at Griewank and Walther's Incremental Adjoint Recursion [20, p. 41] in Algorithm 3.8. If one does not zero the ith adjoint after node i is swept, then after visiting all of node i's successors, $\bar{v}_i$ will have accumulated all necessary contributions and

$$\bar{v}_i = e_\ell \Phi_\ell' \cdots \Phi_{i+1}' e_i. \tag{3.12}$$

In conclusion, we have deduced the same reverse gradient algorithm, once in a graph setting (Algorithm 3.5) and now in an algebraic setting.

Algorithm 3.8: Reverse Gradient Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\bar{v}_{\ell+n} = 1$
for $i = 1, \ldots, \ell$ **do**
  $v_i = \phi_i(v_j)_{j \prec i}$
end
for $i = \ell, \ldots, 1$ **do**
  foreach $j \prec i$ **do**
    $\bar{v}_j += \bar{v}_i \dfrac{\partial \phi_i}{\partial v_j}$
  end
end
Output: $\nabla f(x)^T = \bar{v}^T P^T$

For the Block Forward Gradient Algorithm 3.7, we need an auxiliary matrix $\dot{V} \in \mathbb{R}^{(n+\ell)\times n}$ to accumulate the product sequence of Jacobian matrices. At the end of the ith iteration of Algorithm 3.7,

$$\dot{V} = \Phi_i' \Phi_{i-1}' \cdots \Phi_1' P^T. \tag{3.13}$$

Comparing (3.13) and (3.8), we see that the $(i+n)-$th row of $\dot{V}$ is the gradient ∇v_i^T, thus at the end of ith iteration $\dot{V}$ is given by (3.14).

$$\dot{V} = \begin{bmatrix} 0 \cdots 0 \cdots 0 \\ \vdots \\ 0 \cdots 0 \cdots 0 \\ \nabla v_i^T \\ \vdots \\ \nabla v_{1-n}^T \end{bmatrix}. \tag{3.14}$$

Using once again Φ_i''s special structure we see that $\dot{V} \leftarrow \Phi_i' \dot{V}$ only alters the $(i+n)$-th row of the matrix $\dot{V}$:

$$\dot{V}_{i\bullet} \leftarrow (c_{1-n}^i, \ldots, c_{i-1}^i, 0, \ldots, 0) \cdot \dot{V}. \tag{3.15}$$

Thus the update in (3.15) is equivalent to:

$$\nabla v_i \leftarrow \sum_j c_j^i \nabla v_j. \tag{3.16}$$

With (3.16), Algorithm 3.7 can be translated to Algorithm 3.9 which is the well-known Forward Gradient Algorithm [32] .

Algorithm 3.9: Forward Gradient Algorithm

Input: $x \in \mathbb{R}^n$
Initialization:
$\nabla v_i = e_i, i = 1-n, \ldots, 0$
$\nabla v_i = 0, i = 1, \ldots, \ell$
for $i = 1, \ldots, \ell$ **do**
  $v_i = \phi_i(v_j)_{j \prec i}$
  foreach $j \prec i$ **do**
    $\nabla v_i += \dfrac{\partial \phi_i}{\partial v_j} \nabla v_j$
  end
end
Output: $\nabla f(x) = \nabla v_\ell$

The correctness of the forward and reverse gradient algorithms has been established given that they correctly calculate (3.6). The next natural concern is complexity. It is obvious that the complexity depends on the number of predecessors each ϕ_i is allowed to have. Therefore, to bound the complexity, we will adopt the usual assumption: The elemental functions have at most two arguments. Turning to Algorithm 3.8, the first forward loop has the same complexity as a function evaluation. In each iteration of the reverse sweep, a sum, a multiplication and elemental function evaluation are performed for each predecessor. Since there are at most two predecessors, each iteration of the revere sweep is bounded by

a constant. Hence the reverse sweep has a complexity of $O(\ell)$ which is a fundamental result in automatic differentiation:

$$\text{TIME}(eval(\nabla f(x))) = \text{TIME}(eval(f(x))).$$

This result is referred to as the "Cheap Gradient Principle" [20]. The remaining (and equally important) aspect regarding complexity is memory usage. For all reverse mode algorithms have to store information in the prior forward sweep to be able to carry out the reverse sweep. In some cases all the floating point values v_i, for $i = 1, \ldots, \ell$, have to be recorded, and the list of intermediate variables is potentially very long. We will address these issues in Chapter 5 .

The Forward Gradient Algorithm 3.9 is not so cheap, for it operates on vectors of n-dimensions. In each iteration, a vector sum, a scalar-vector multiplication and n elemental function evaluations are performed. This yields a complexity of $n\text{TIME}(eval(f(x)))$. The advantage is that there is no issue with memory usage.

Algorithm 3.10: Block Mixed Mode Gradient Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\dot{V} \leftarrow P^T \in \mathbb{R}^{(n+\ell)\times n}$, $\bar{v} \leftarrow e_{\ell+n}$
(*forward sweep to calculate each* Φ_i)
for $i = 1, \ldots, p$ **do**
    $\dot{V} \leftarrow \Phi_i' \dot{V}$
end
for $i = \ell, \ldots, p+1$ **do**
    $\bar{v}^T \leftarrow \bar{v}^T \Phi_i'$
end
Output: $\nabla f(x)^T = \bar{v}^T \dot{V}$

Though both the Forward and Reverse Gradient Algorithms can be easily demonstrated using induction instead of the state transformation perspective (3.6), this formula allows us to investigate other possibilities. For instance, are these the only two plausible algorithms for calculating the gradient? Restricting our attention to algorithms that perform at most, a forward sweep followed by a reverse sweep, what are the possibilities? Accumulating from right to left up to the p-th Jacobian matrix, and from left to right up to $(p+1)$-th Jacobian matrix, gives us a Block Mixed Mode Gradient Algorithm 3.10. An immediate advantage of this mixed mode algorithm is that it involves two disjoint set of operations. Hence the work load can be separated over two processors.

If the gradient can be evaluated at a cost proportional to the function evaluation, how cheaply can the Hessian matrix be evaluated? This is the subject of the next chapter, and basically the entire dissertation.

Chapter 4

Calculating the Hessian Matrix

4.1 Calculating The Hessian Using The Computational Graph

4.1.1 Computational Graph of the Gradient

Creating a graph model for the Hessian is also very useful, as it provides insight and intuition regarding the workings of Hessian algorithms. Not only can the graph model suggest algorithms, it can also be very enlightening to interpret the operations performed by an algorithm as operations on variables associated with the nodes and arcs of a computational graph.

Since second order derivatives are simply first order derivatives of the gradient, a natural approach to its calculation would be to build a computational graph $G^g = (V^g, E^g)$ for the gradient evaluation in Algorithm 3.5, then use the formula (3.3) for partial derivatives expressed in a graph setting on G^g to obtain the second order partial derivatives. Later on we shall be able to make do without the enlarged graph, but it helps in formulating the problem.

Let $\mathcal{CG} = (G, \varphi)$ be the computational graph of the function $f(x)$ where $G = (V \cup Z, E)$. Of course the gradient may be represented by distinct computational graphs, or equivalently, sequential lists of functions, but the natural one to consider is the one associated with the computation performed by the Reverse Gradient Algorithm 3.5. Assuming this choice, the gradient $\nabla f = (\bar{v}_{1-n}, \ldots, \bar{v}_0)^T$ is a composite function of $(\bar{v}_1, \ldots, \bar{v}_\ell)$, as well as $(v_{1-n}, \ldots, v_\ell)$, which implies that the gradient (computational) graph $G^g = (V^g, E^g)$ must contain G. The graph G^g is basically built upon G by adding nodes associated with $\bar{v}_i$, for $i = 1-n, \ldots, \ell$, and edges representing the functional dependencies amongst these nodes.

Thus the node set V^g contains $2(n+\ell)$ nodes $\{1-n, \ldots, \ell, \overline{1-n}, \ldots, \bar{\ell}\}$, the first half associated with the original variables and the second half with the adjoint variables. The arc set is $E^g = E_1 \cup E_2 \cup N$, where E_1 contains arcs with both endpoints in "original" nodes, E_2 arcs with both endpoints in "adjoint" nodes and N, arcs with endpoints of mixed nature. Since running Algorithm 3.5 does not introduce new dependencies amongst the original v's,

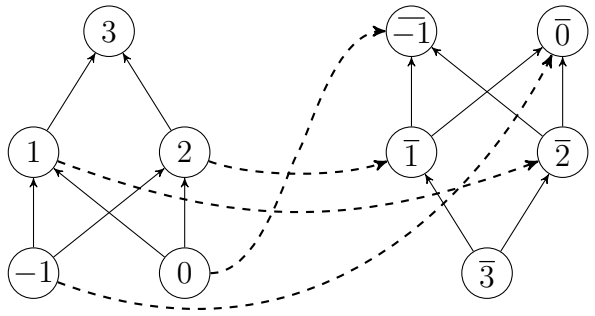
we have that $E_1 = E$.

The new dependent variables created by running Algorithm 3.5 satisfy

$$\bar{v}_i = \sum_{k|i\prec k} \bar{v}_k \frac{\partial \phi_k}{\partial v_i}(v_{P(k)}) \tag{4.1}$$

at the end of the algorithm.

Expression (4.1) indicates that $\bar{v}_i$ depends on $\bar{v}_k$ for every k that is a successor of i. Thus every arc $(i,k) \in E_1$ gives rise to arc $(\bar{k},\bar{\imath}) \in E_2$. Therefore, arcs in E_2 are copies of the arcs in E with the orientation reversed. The graph G^g thus contains G and a kind of a mirror copy of G. Furthermore, $j \prec \bar{\imath}$ only if v_j is an input variable of the function $\partial\phi_k/\partial v_i$. Notice that this may only happen if j and i share a common successor k. This implies, in particular, that there are no edges incident to $\bar{\ell}$. Figure 4.1 shows the computational graph of the gradient of the function $f(x) = (x_{-1}x_0)(x_{-1} + x_0)$. On the left we have the computational graph of f and, on the right, a mirror copy thereof. Arcs in N are the ones drawn dashed in the picture. Such a gradient computational graph has already been obtained in [20, p. 237].



$$\begin{array}{l|l} v_{-1} = x_{-1} & \bar{v}_3 = \phi_{\overline{3}} = 1 \\ v_0 = x_0 & \bar{v}_2 = \phi_{\overline{2}} = \bar{v}_3 v_1 \\ v_1 = \phi_1 = v_{-1}v_0 & \bar{v}_1 = \phi_{\overline{1}} = \bar{v}_3 v_2 \\ v_2 = \phi_2 = v_{-1} + v_0 & \bar{v}_0 = \phi_{\overline{0}} = \bar{v}_2 + \bar{v}_1 v_{-1} \\ v_3 = \phi_3 = v_1 v_2 & \bar{v}_{-1} = \phi_{\overline{-1}} = \bar{v}_2 + \bar{v}_1 v_0 \end{array}$$

Figure 4.1: Gradient computational graph G^g of the function $f(x) = (x_{-1}x_0)(x_{-1} + x_0)$.

Analogously to (3.3), we conclude that

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{p|\text{path from } i \text{ to } \bar{\jmath}} \text{weight of path } p, \tag{4.2}$$

where again the weight of path p is simply the product of the weights of the arcs in p.

The weights of arcs $(i,k) \in E_1$ are already know. Equation (4.1) implies that the weight of arc $(\bar{k},\bar{\imath}) \in E_2$ is

$$c^{\bar{k}}_{\bar{\imath}} = \frac{\partial \bar{v}_i}{\partial \bar{v}_k} = \frac{\partial \phi_i}{\partial v_k} = c^i_k, \tag{4.3}$$

that is, arc (i,k) has the same weight as its mirror image.

The weight of arc $(j, \bar{\imath})$ is also obtained from (4.1)

$$\begin{aligned} c_j^{\bar{\imath}} &= \sum_{k|i\prec k} \bar{v}_k \frac{\partial^2 \phi_k}{\partial v_j \partial v_i} \\ &= \sum_{k|i\prec k \text{ and } j\prec k} \bar{v}_k \frac{\partial^2 \phi_k}{\partial v_j \partial v_i}, \end{aligned} \tag{4.4}$$

since the partial derivative $\partial^2\phi_k/\partial v_j \partial v_i$ is identically zero if k is not a successor of j. In particular, (4.4) and the fact that f is twice continuously differentiable implies that

$$c_j^{\bar{\imath}} = c_i^{\bar{\jmath}}, \text{ for } j \neq i. \tag{4.5}$$

Notice that arcs in N are the only ones with second-order derivatives as weights. In a sense, they carry the nonlinearity of f, which suggests the denomination *nonlinear arcs*.

4.1.2 Computation of the Hessian

The computation of a second order partial derivative could be accomplished by applying existing Jacobian methods to the gradient graph G^g, which has been done in [3]. Instead, we wish to somehow include in G the pertinent information from G^g and develop rules to deal with the new information that make it possible to evaluate all entries of the Hessian in one backward sweep of the graph.

The basic idea is to identify paths in G^g using the unique nonlinear arc in the path. Unique, for none of the original nodes is a successor of an adjoint node. Therefore, the summation in (4.2) may be partitioned as follows:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{(r,\bar{s})\in N} \left[\left(\sum_{p|\text{path from } i \text{ to } r} \text{weight of path } p \right) c_r^{\bar{s}} \left(\sum_{q|\text{path from } \bar{s} \text{ to } \bar{\jmath}} \text{weight of path } q \right) \right]. \tag{4.6}$$

On close examination, there is a lot of redundant information in G^g. One really doesn't need the mirror copy of G, since the information attached to the adjoint nodes can be recorded associated to the original nodes. Now if we fold back the mirror copy over the original, identifying nodes k and $\overline{k}$, we obtain a graph with same node set as G but with an enlarged set of arcs. Arcs in E_1 will be replaced by pairs of arcs in opposite directions and nonlinear arcs will become either loops (in case one had an arc $(i, \bar{\imath})$ in N) or pairs of arcs with opposite orientations between the same pair of nodes, see Figure 4.2. Due to the symmetry in (4.3), the arc weights of the mirror arcs are the same, therefore we will try to make do without these mirror copies. Thus we arrive at a simplified graph which is in fact the original function graph $G = (V \cup Z, E)$ with added nonlinear edges N, see the rightmost graph in Figure 4.2. We will denote this enlarged graph with additional nonlinear edges by $G^N = (V \cup Z, E \cup N)$.

The paths needed for the computation of the Hessian, in G^N, are divided into three parts. In the first part we have a directed path from some zero in-degree node, say i, to some other node, say r. Next comes a nonlinear arc (r, s). The last part is a path from s to another zero in-degree node, say j, in which all arcs are traveled in the wrong direction. Of course, both the first and third parts of the path may be empty, only the middle part (the nonlinear arc) is mandatory. Thus we have:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{(r,s)\in N} \left[\left(\sum_{p|\text{path from } i \text{ to } r} \text{weight of path } p \right) c_r^s \left(\sum_{q|\text{path from } j \text{ to } s} \text{weight of path } q \right) \right], \tag{4.7}$$

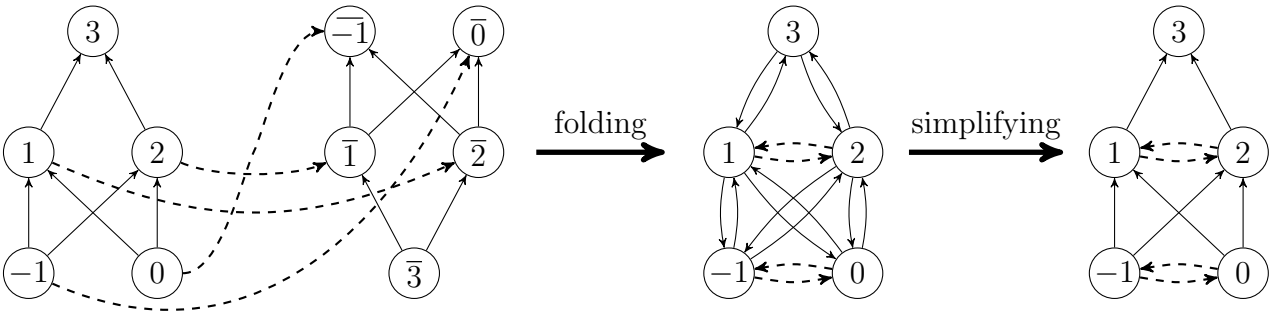


Figure 4.2: Folding of the gradient computational graph of Figure 4.1 and further elimination of redundancies leaves us with the graph G^N.

We have now expressed an arbitrary second order partial derivative in terms of a sum of the weights of certain paths in G^N. To devise an efficient way to accumulate the contribution of all these paths, we grow these paths from the nonlinear edges. This is more efficient, as opposed to growing from the independent nodes, for the paths that contain a nonlinear edge (r, s) may start at distinct endpoints, but all funnel in as they approach the nonlinear edge. The key idea is the creation of shortcuts, as exemplified in Figure 4.3. All paths that contain the edge $(r, s) \in N$ must pass through a predecessor of r. To illustrate, let $P(r) = \{i, j\}$. All paths containing (r, s) must contain the sub-path (i, r, s) or (j, r, s). Therefore, (r, s) may be eliminated and replaced by two new artificial shortcuts (i, s) and (j, s) with weights $c_i^r c_r^s$ and $c_j^r c_r^s$ respectively. What if G^N already contained one of these new nonlinear edges? In this case, we do a parallel reduction, that is, replace the parallel edges (original plus shortcut) with one edge whose weight is the sum of the weights of the parallel edges. This process is repeated until the resulting shortcut edges connect zero in-degree nodes, such that the weight of arc $(i - n, j - n)$ is the sum of the weights of all permissible paths between these nodes. We call this process `Pushing`, for the contribution of nonlinear edges are pushed down to predecessors.

A simple algorithm for calculating (4.7), is to store the sum of the weights of all permissible paths between nodes j and k in an edge denoted by w_{jk}. One would first add the shortest permissible paths to $G = (V \cup Z, E)$, thoughs consisting of a single nonlinear arc,

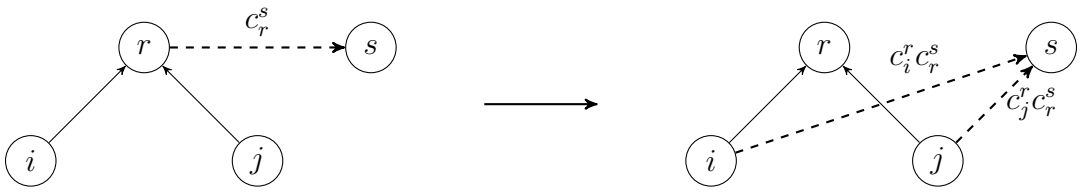


Figure 4.3: Pushing the edge (r, s)

thus producing $G^N = (V \cup Z, E \cup N)$. Then recursively create shortcuts using the `Pushing` routine, until the only shortcut edges left connect independent nodes, see Algorithm 4.1. We have named this algorithm `edge_pushing` after the pushing subprocedure.

Our objective here has been to develop intuition on how one should accumulate the second order partial derivatives. There is still more symmetry to explore, for instance, equation (4.5) implies that nonlinear arcs in parallel have the same weight. Thus we could replace the pairs of directed nonlinear arcs in opposite directions with a single undirected arc and directed loops by undirected ones, and see what this implies for the calculation of (4.7). Alas this has some awkward consequences that would only hinder our intuition, though further on, we present and deduce a more complete and implemented version of `edge_pushing` that takes full advantage of all these symmetries and of sparsity.

4.2 Calculating the Hessian Using State Transformations

In a fashion similar to the gradient formula (3.6), we develop a Hessian formula that we will use to demonstrate several algorithms encountered in the literature. We also use this formula to design a new forward Hessian algorithm and a new reverse Hessian algorithm called `edge_pushing`. Our approach will be analogous to that used to deduce gradient methods using state transformations: first we describe a Hessian Algorithm in a Block format and then translate this algorithm to a componentwise format. More attention will be given to the `edge_pushing` algorithm, for it is our chosen method to implement and test.

The closed formula to be developed concerns the Hessian of a function g that is defined as a linear combination of the functions $\Psi_1, \ldots, \Psi_p$, or, in matrix form,

$$g(x) = y^T \Psi(x), \tag{4.8}$$

where $y \in \mathbb{R}^p$ and $\Psi \in C^2(\mathbb{R}^n, \mathbb{R}^p)$. The linearity of the differential operator implies that the Hessian of g is simply the linear combination of the Hessians of $\Psi_1, \ldots, \Psi_p$:

$$g''(x) = \sum_{i=1}^{p} y_i \Psi_i''(x). \tag{4.9}$$

Algorithm 4.1: Simple `edge_pushing`

Input: $G = (V \cup Z, E)$, $|V| = \ell$, $\{\phi_i \mid i = 1, \ldots, \ell\}$, $\{\bar{v}_i \mid i = 1 \ldots \ell\}$
Initialization $W = \{\emptyset\}$
`Creating Paths with one edge`:
for $i = 1, \ldots, \ell$ **do**
  foreach *nonlinear* ϕ_i *and* $r, s \in P(i)$ **do**
    $w_{rs} \mathrel{+}= \bar{v}_i \dfrac{\partial^2 \phi_i}{\partial v_r \partial v_s}$
  end
end
`Pushing`:
foreach $(r, s) \in W$ **do**
  if $r \notin Z$ **then**
    foreach $j \in P(r)$ **do**
      $w_{js} \mathrel{+}= \dfrac{\partial \phi_r}{\partial v_j} w_{rs}$
    end
  else if $k \notin Z$ **then**
    foreach $j \in P(s)$ **do**
      $w_{rj} \mathrel{+}= \dfrac{\partial \phi_s}{\partial v_j} w_{rs}$
    end
  Delete (r, s)
end
Output: Edge set W

This motivates the introduction of the following definition of the vector-tensor product $y^T\Psi''(x)$, in order to establish an analogy between the linear combinations in (4.8) and (4.9):

$$g''(x) = (y^T\Psi(x))'' = y^T\Psi'' = \sum_{i=1}^{p} y_i\Psi_i''(x). \tag{4.10}$$

Next we need to establish how to express g'' when Ψ is a composition of vector functions of several variables, the subject of the next Proposition.

Proposition 4.1 *Let $y \in \mathbb{R}^p$, $\Omega \in C^2(\mathbb{R}^n, \mathbb{R}^m)$, $\Theta \in C^2(\mathbb{R}^m, \mathbb{R}^p)$ and $\Psi(x) = \Theta \circ \Omega(x)$. Then*

$$y^T\Psi'' = (\Omega')^T(y^T\Theta'')\Omega' + (y^T\Theta')\Omega''. \tag{4.11}$$

Demonstration: By definition, applying differentiation rules, and using the symmetry of the Hessian, we may calculate entry (j,k) of the Hessian as follows:

$$\begin{aligned}
(y^T\Psi''(x))_{jk} &= \sum_i y_i \frac{\partial^2\Psi_i(x)}{\partial x_j \partial x_k} \\
&= \sum_i y_i \frac{\partial}{\partial x_j}\left(\frac{\partial\Theta_i(\Omega(x))}{\partial x_k}\right) \\
&= \sum_i y_i \frac{\partial}{\partial x_j}\left(\sum_{r=1}^{m} \frac{\partial\Theta_i(\Omega(x))}{\partial\Omega_r}\frac{\partial\Omega_r(x)}{\partial x_k}\right) \\
&= \sum_i\sum_r y_i \left[\frac{\partial}{\partial x_j}\left(\frac{\partial\Theta_i(\Omega(x))}{\partial\Omega_r}\right)\right]\frac{\partial\Omega_r(x)}{\partial x_k} + \sum_i\sum_r y_i \frac{\partial\Theta_i(\Omega(x))}{\partial\Omega_r}\frac{\partial^2\Omega_r(x)}{\partial x_j\partial x_k} \\
&= \sum_r\sum_s\sum_i y_i \frac{\partial^2\Theta_i(\Omega(x))}{\partial\Omega_s\partial\Omega_r}\frac{\partial\Omega_s(x)}{\partial x_j}\frac{\partial\Omega_r(x)}{\partial x_k} + \sum_r (y^T\Theta'(\Omega(x))_r(\Omega_r''(x))_{jk} \\
&= \sum_s\sum_r (y^T\Theta''(\Omega(x)))_{rs}\,(\Omega'(x))_{sj}\,(\Omega'(x))_{rk} + \sum_r (y^T\Theta'(\Omega(x))_r(\Omega_r''(x))_{jk} \\
&= \sum_s (\Omega'(x))_{sj}\sum_r (y^T\Theta''(\Omega(x)))_{sr}\,(\Omega'(x))_{rk} + \sum_r (y^T\Theta'(\Omega(x))_r(\Omega_r''(x))_{jk}, \\
&= \left((\Omega(x))^T(y^T\Theta''(\Omega(x)))\Omega'(x)\right)_{jk} + \left((y^T\Theta'(\Omega(x)))\Omega''(x)\right)_{jk},
\end{aligned}$$

which is the entry (j,k) of the right-hand-side of (4.11). ∎

Although we want to express the Hessian of a composition of state transformations, it is actually easier to obtain the closed form for the composition of generic vector multivariable functions, our next result.

Proposition 4.2 *Let $\Psi_i(x) \in C^2(\mathbb{R}^{m_{i-1}}, \mathbb{R}^m)$, for $i = 1, \ldots, k$, $y \in \mathbb{R}^{m_k}$ and*

$$g(x) = y^T\Psi_k \circ \cdots \circ \Psi_1(x).$$

Then

$$g'' = \sum_{i=1}^{k} \left(\prod_{j=1}^{i-1} (\Psi'_j)^T \right) ((\overline{\mathrm{w}}^i)^T \Psi''_i) \left(\prod_{j=1}^{i-1} \Psi'_{i-j} \right), \tag{4.12}$$

where

$$(\overline{\mathrm{w}}^i)^T = y^T \prod_{j=1}^{k-i} \Psi'_{k-j+1}, \quad \textit{for } i = 1, \ldots, k. \tag{4.13}$$

Demonstration: The proof is by induction on k. When $k = 1$, the result is trivially true, since in this case (4.12)–(4.13) reduce to $(\overline{\mathrm{w}}^1)^T \Psi''_1 = y^T \Psi''_1$, which denotes, according to (4.9), the Hessian of g.

Assume the proposition is true when g is the composition of $k-1$ functions. Now simply rewrite the composition of k functions as follows

$$g = y^T \Psi_k \circ \cdots \circ \Psi_3 \circ \Psi, \tag{4.14}$$

where $\Psi = \Psi_2 \circ \Psi_1$. Then, applying the induction hypothesis to (4.14), we obtain

$$g'' = (\Psi')^T \left[\sum_{i=3}^{k} \left(\prod_{j=3}^{i-1} (\Psi'_j)^T \right) ((\overline{\mathrm{w}}^i)^T \Psi''_i) \left(\prod_{j=3}^{i-1} \Psi'_{i-j} \right) \right] \Psi' + (\overline{\mathrm{w}}^2)^T \Psi''. \tag{4.15}$$

The last term in (4.15) is calculated separetely, using the induction hypothesis, (4.11) and (4.13):

$$\begin{aligned} (\overline{\mathrm{w}}^2)^T \Psi'' &= (\Psi'_1)^T ((\overline{\mathrm{w}}^2)^T \Psi''_2) \Psi'_1 + ((\overline{\mathrm{w}}^2)^T \Psi'_2) \Psi''_1 \\ &= (\Psi'_1)^T ((\overline{\mathrm{w}}^2)^T \Psi''_2) \Psi'_1 + (\overline{\mathrm{w}}^1)^T \Psi''_1. \end{aligned} \tag{4.16}$$

Using the fact that $\Psi' = \Psi'_2 \Psi'_1$, and expression (4.16) obtained for the last term, (4.15) becomes

$$\begin{aligned} g'' &= (\Psi'_1)^T (\Psi'_2)^T \left[\sum_{i=3}^{k} \left(\prod_{j=3}^{i-1} (\Psi'_j)^T \right) ((\overline{\mathrm{w}}^i)^T \Psi''_i) \left(\prod_{j=3}^{i-1} \Psi'_{i-j} \right) \right] \Psi'_2 \Psi'_1 \\ &\quad + (\Psi'_1)^T ((\overline{\mathrm{w}}^2)^T \Psi''_2) \Psi'_1 + (\overline{\mathrm{w}}^1)^T \Psi''_1 \\ &= \sum_{i=1}^{k} \left(\prod_{j=1}^{i-1} (\Psi'_j)^T \right) ((\overline{\mathrm{w}}^i)^T \Psi''_i) \left(\prod_{j=1}^{i-1} \Psi'_{i-j} \right), \end{aligned}$$

which completes the proof. ∎

The Hessian of the composition of state transformations follows easily from Proposition 4.2.

Corollary 4.1 *Let f be the composition of state transformations given in (3.5). Then its Hessian is*

$$f'' = P \sum_{i=1}^{\ell} \left(\prod_{j=1}^{i-1} (\Phi_j')^T \right) ((\overline{\mathrm{v}}^i)^T \Phi_i'') \left(\prod_{j=1}^{i-1} \Phi_{i-j}' \right) P^T, \tag{4.17}$$

where

$$(\overline{\mathrm{v}}^i)^T = e_\ell^T \prod_{j=1}^{\ell-i} \Phi_{\ell-j+1}', \quad \textit{for } i = 1, \ldots, \ell. \tag{4.18}$$

Demonstration: Simply apply (4.12) to the composition of $\ell+1$ functions, where $\Psi_i = \Phi_i$, for $i = 1, \ldots, \ell$, $\Psi_0(x) = P^T x$, and use the facts that $\Psi_0' = P^T$ and $\Psi_0'' = 0$. ∎

The expression for the Hessian of f can be further simplified by noting that the tensor Φ_i'' is null except for the Hessian of its component i, $[\Phi_i]_i$, since the other components are just projections onto a single variable, see (3.9). Thus the vector-tensor product in (4.17) reduces to

$$(\overline{\mathrm{v}}^i)^T \Phi_i'' = \overline{\mathrm{v}}_i^i [\Phi_i]_i'', \tag{4.19}$$

where $\overline{\mathrm{v}}^i = (\overline{\mathrm{v}}_{1-n}^i, \ldots, \overline{\mathrm{v}}_\ell^i)$. Additionally, each vector $\overline{\mathrm{v}}^i$ defined in (4.18) is equivalent to the vector $\overline{v}$ in Algorithm 3.6 immediately after iteration i. Turning now to Algorithm 3.8, we see that $\overline{\mathrm{v}}_i^i$ is simply the ith adjoint $\bar{v}_i$. Furthermore, notice that $[\Phi_i]_i''$ is just the Hessian of ϕ_i padded with the appropriate number of zeros at the right places:

$$[\Phi_i]_i'' = \left(\frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \right)_{j=1-n \ldots \ell}^{k=1-n \ldots \ell}.$$

Letting

$$\dot{V}^i = \left(\prod_{j=1}^{i-1} \Phi_{i-j}' \right) P^T, \quad \text{for } i = 1, \ldots, \ell, \tag{4.20}$$

and using (4.19), (4.17) reduces to

$$\begin{aligned} f'' &= \sum_{i=1}^{\ell} (\dot{V}^i)^T \bar{v}_i [\Phi_i]_i'' \dot{V}^i \\ &= \sum_{i=1}^{\ell} \bar{v}_i (\dot{V}^i)^T [\Phi_i]_i'' \dot{V}^i. \end{aligned} \tag{4.21}$$

4.2.1 A New Forward Hessian Algorithm

If one first calculates and stores the adjoint values $\bar{v}_i$, $i = 1, \ldots, \ell$, then the formula (4.21) can be naturally calculated in a forward sweep, see Algorithm 4.2. In each iteration, a summand of (4.21) is calculated and added to the matrix W. Note that at the end of iteration $i+1$ of the forward sweep

$$\dot{V} = \Phi_i' \cdots \Phi_1' P^T,$$

therefore $\dot{V} = \dot{V}^{i+1}$. At the end of the ℓ-th iteration of the forward sweep, all ℓ summands of (4.21) will have been added to the matrix W, hence $W = f''$.

Algorithm 4.2: Block Forward Hessian Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\overline{v} = e_\ell$, $W = 0 \in \mathbb{R}^{n \times n}$, $\dot{V} = P^T$
for $i = \ell, \ldots, 1$ **do**
  $\bar{v}_i = \sum_{i \prec j} \bar{v}_j \frac{\partial \phi_j}{\partial v_i}$
end
for $i = 1, \ldots, \ell$ **do**
  $W = W + \bar{v}_i (\dot{V})^T [\Phi_i]_i'' \dot{V}$
  $\dot{V} = \Phi_i' \dot{V}$
end
Output: $f''(x) = W$

Using (3.14), at the end of the ith iteration of the forward sweep in Algorithm 4.2, one can break $\dot{V}$ up into gradients of the intermediate variables and see the following equivalence:

$$(\dot{V})^T [\Phi_i]_i'' \dot{V} = \sum_{j,k} \nabla v_j \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \nabla v_k^T. \tag{4.22}$$

Finally, by noting that the update $\dot{V} = \Phi_i' \dot{V}$ is merely an iteration of the Forward Gradient Algorithm 3.9, this Forward Hessian Algorithm 4.2 can be rewritten as Algorithm 4.3.

To bound the complexity of Algorithm 4.3, let us assume once more that each node has at most two predecessores. The Forward Hessian Algorithm 4.3 performs all the calculations of the Forward and Reverse Gradient Algorithms 3.9 and 3.8, which have a complexity of $O(n\ell)$ and $O(\ell)$ respectively. In addition, for every nonlinear function it calculates an outer product matrix, a computation that costs $O(n^2)$. Thus the complexity is bound by:

$$O\left(n\ell + n^2(\text{Number of nonlinear intermediate functions})\right). \tag{4.23}$$

In terms of memory use, the Forward Hessian must store all ℓ adjoints in a reverse sweep. While in the forward sweep it carries a $n \times n$ matrix W and ℓ vectors of size n, thus using up an extra $O(n\ell)$, though this may be improved by using sparsity and overwriting schemes. An advantage of the algorithm is that it preserves the symmetry of W, a desirable quality. We have found no reference to such an algorithm in the literature.

4.2.2 Intermediate Forward Hessian Algorithm

Another strategy for calculating the Hessian matrix is to adopt the viewpoint of the intermediate variables as functions of the independent variables (3.7), then establish a recurrence

Algorithm 4.3: Forward Hessian Algorithm

Input: $x \in \mathbb{R}^n$
Initialization: $\overline{v} = e_\ell$, $W = 0 \in \mathbb{R}^{n \times n}$
$\nabla v_i = e_i, i = 1-n, \ldots, 0$
$\nabla v_i = 0, i = 1, \ldots, \ell$
for $i = \ell, \ldots, 1$ **do**
  $\bar{v}_i = \sum_{i \prec j} \bar{v}_j \frac{\partial \phi_j}{\partial v_i}$
end
for $i = 1, \ldots, \ell$ **do**
  foreach $j, k \prec i$ **do**
    $W \mathrel{+}= \bar{v}_i \nabla v_j \dfrac{\partial^2 \phi_i}{\partial v_j \partial v_k} \nabla v_k^T$
  end
  foreach $j \prec i$ **do**
    $\nabla v_i \mathrel{+}= \dfrac{\partial \phi_i}{\partial v_j} \nabla v_j$
  end
end
Output: $f''(x) = W$

that involves the Hessian of $v_i(x)$ and the Hessians of its predecessors. One can then calculate the Hessians of the intermediate variables in a forward sweep, up to and including the Hessian of $v_\ell(x) = f(x)$. This is perhaps the most common method found in the literature, apparently appearing first in Jackson and McCormick's work [23]. There are also a number of other references [1] and Griewank and Walther's book [20, p.155]. This is a method analogous to the Forward Gradient Algorithm 3.9, where the gradient of each intermediate variable is calculated using the gradients of predecessors.

The demonstration presented here uses (4.21) and is admittedly cumbersome. The usual demonstration using induction is more natural. This being said, we present this demonstration for reasons of completion.

To demonstrate this method, we define adjoints associated to the calculation of each intermediate variable $v_j(x)$. For this we use an analogous formula to (3.12):

$$\begin{aligned} \bar{v}_i^j &= e_j^T \Phi_j' \cdots \Phi_{i+1}' e_i, \quad \text{for } i = 1, \ldots, j-1, \\ \bar{v}_j^j &= 1, \\ \bar{v}_i^j &= 0, \qquad\qquad\quad \text{for } i > j. \end{aligned} \tag{4.24}$$

With this notation, one can establish a recurrence relation between adjoint variables:

$$\begin{aligned}
\bar{v}_i^j &= e_j^T \Phi'_j \cdots \Phi'_{i+1} e_i \\
&= \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} e_k^T \Phi'_{j-1} \cdots \Phi'_{i+1} e_i \\
&= \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} e_k^T \Phi'_k \cdots \Phi'_{i+1} e_i \\
&= \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} \bar{v}_i^k.
\end{aligned}$$

The last but one equality holds for $e_k^T \Phi'_i = e_k^T$ when $i > k$. With this, one can use the Hessian formula (4.21) to express the Hessian of $v_j(x)$:

$$\begin{aligned}
v''_j(x) &= \sum_{i=1}^{j} \bar{v}_i^j (\dot{V}^i)^T [\Phi_i]''_i \dot{V}^i \\
&= \sum_{i=1}^{j-1} \bar{v}_i^j (\dot{V}^i)^T [\Phi_i]''_i \dot{V}^i + \bar{v}_j^j (\dot{V}^j)^T [\Phi_j]''_j \dot{V}^j \\
&= \sum_{i=1}^{j-1} \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} \bar{v}_i^k (\dot{V}^i)^T [\Phi_i]''_i \dot{V}^i + 1 (\dot{V}^j)^T [\Phi_j]''_j \dot{V}^j \\
&= \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} \sum_{i=1}^{k} \bar{v}_i^k (\dot{V}^i)^T [\Phi_i]''_i \dot{V}^i + (\dot{V}^j)^T [\Phi_j]''_j \dot{V}^j \\
&= \sum_{k \prec j} \frac{\partial \phi_j}{\partial v_k} v''_k(x) + (\dot{V}^j)^T [\Phi_j]''_j \dot{V}^j. \qquad (4.25)
\end{aligned}$$

Equation (4.25) establishes the desired recurrence relation. Using (4.22) once again and the recurrence relation (4.25), one can calculate the Hessian matrices $v''_i(x)$, $i = 1, \ldots, \ell$, in a forward sweep using Algorithm 4.4.

This Intermediate Forward Hessian Algorithm 4.4 is the only Hessian algorithm that performs all necessary calculations in a single forward sweep. Though this is a significant advantage, the downside to this algorithm is that it potentially needs a large quantity of memory with ℓ Hessian $n \times n$ matrices. However, exploiting the symmetry of the Hessians, this quantity is almost halved, as is normally done [1]. The time complexity of Algorithm 4.4 is dominated by the calculation of the matrix outer-products and the gradients of the intermediate variables, and thus has the same complexity as the Forward Hessian, see (4.23). If one is interested in sensitivity issues of the function with respect to its intermediate variables, then Algorithm 4.4 may be useful. Otherwise, if one is solely interested in the Hessian of the function $f(x)$, then the large storage requirements may render this algorithm prohibitive for many applications.

Algorithm 4.4: Intermediate Forward Hessian

Input: $x \in \mathbb{R}^n$
Initialization: $v_i''(x) = 0 \in \mathbb{R}^{n \times n}$, for $i = 1, \ldots, \ell$
$\nabla v_i = e_i, i = 1 - n, \ldots, 0$
$\nabla v_i = 0, i = 1, \ldots, \ell$
for $i = 1, \ldots, \ell$ **do**
  $v_i''(x) = \sum_{j \prec i} \frac{\partial \phi_i}{\partial v_j} v_i''(x)$
  foreach $j, k \prec i$ **do**
    $v_i''(x) += \nabla v_j \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \nabla v_k^T$
  end
  foreach $j \prec i$ **do**
    $\nabla v_i += \frac{\partial \phi_i}{\partial v_j} \nabla v_j$
  end
end
Output: $f''(x) = v_\ell''(x)$

4.2.3 Griewank and Walther's Reverse Hessian Algorithm

Although the inception of Griewank and Walther's reverse Hessian computation algorithm [20, p.157], presented in block form in Algorithm 4.5, follows a different line of reasoning, its correctness may be established by means of (4.17).

Algorithm 4.5: Griewank and Walther's Reverse Hessian computation algorithm.

Input: $x \in \mathbb{R}^n$
Initialization: $\overline{v} = e_\ell$, $W = 0$, $\dot{V}^1 = P^T$
for $i = 2, \ldots, \ell$ **do**
  $\dot{V}^i = \Phi_{i-1}' \dot{V}^{i-1}$
end
for $i = \ell, \ldots, 1$ **do**
  $W = (\Phi_i')^T W$
  $W += \overline{v}^T \Phi_i'' \dot{V}^i$
  $\overline{v}^T = \overline{v}^T \Phi_i'$
end
Output: $f'' = PW$

Algorithm 4.5 is the translation to block operations, using our notation, of Griewank and Walther's reverse Hessian computation algorithm. It recursively builds parts of expression

(4.17) and then combines them appropriately. It is straightforward to see that $(\dot{V}^1, \ldots, \dot{V}^\ell)$ constructed in the first (forward) loop satisfies (4.20).

In the second loop the indices are visited in reverse order, or equivalently, a backward sweep of the computational graph is performed. Notice that the computation of $\overline{v}$ is the same as the Reverse Matrix Gradient in Algorithm 3.6. Thus at the beginning of the iteration where node i is swept, this vector contains the partial product $e_\ell^T \Phi'_\ell \cdots \Phi'_{i-1}$. Hence, at the iteration where node i is swept, W is incremented by

$$\overline{v}^T \Phi''_i \dot{V}^i = ((\overline{\mathrm{v}}^i)^T \Phi''_i) \left(\prod_{j=1}^{i-1} \Phi'_{i-j} \right) P^T.$$

Finally, taking into account the pre-multiplication done at the beginning of the reverse loop, it can be shown by induction that, at the end of the iteration where node i is swept, we have

$$W = \sum_{k=i}^{\ell} \left(\prod_{j=i}^{k-1} (\Phi'_j)^T \right) ((\overline{\mathrm{v}}^k)^T \Phi''_k) \left(\prod_{j=1}^{k-1} \Phi'_{k-j} \right) P^T.$$

This implies that, at the end of the algorithm, PW is precisely the expression for the Hessian of f in (4.17).

As far as we can ascertain, there are no reports on the implementation and testing of this algorithm. Although the special structure of the Jacobians and Hessians of the state transformations lead to simple and efficient componentwise versions of the block assignments, there are two obvious downsides to this approach for calculating the Hessian. First is the fact that its symmetry is not exploited, and second, $(\dot{V}^1, \ldots, \dot{V}^\ell)$, calculated in the forward loop, needs to be recorded for later use in the second loop, which is potentially a large quantity of memory, even if one takes advantage of its special structure.

4.2.4 Hessian-vector Product Algorithm

With a slight ajustment to Algorithm 4.5 one can efficiently calculate Hessian-vector products. Let $y \in \mathbb{R}^n$ be a given vector, and $f''(x)y$ the desired Hessian-vector product. We seek an algorithm that calculates (4.17) but with an additional y multplied on the right. Letting

$$\dot{v}^i := \dot{V}^i y = \left(\prod_{j=1}^{i-1} \Phi'_{i-j} \right) P^T y, \quad \text{for } i = 1, \ldots, \ell, \tag{4.26}$$

we can apply a completely analogous strategy to the previous Algorithm 4.5 and calculate the Hessian-vector product by simply swapping $\dot{V}^i$ in Algorithm 4.5 for $\dot{v}^i$ resulting in Algorithm 4.6. This algorithm first appeared in [8].

This Hessian-vector algorithm is renowned for its time complexity which is $O(\ell)$, the same as the function evaluation's time complexity. To see this, we must write Algorithm 4.6

Algorithm 4.6: Block Reverse Hessian-vector

Input: $x \in \mathbb{R}^n$
Initialization $\bar{v} = e_\ell$, $z = 0 \in \mathbb{R}^{(n+\ell)}$, $\dot{v}^1 = P^T y$
for $i = 1, \ldots, \ell$ **do**
  $\dot{v}^i = \Phi_i' \dot{v}^{i-1}$
end
for $i = \ell, \ldots, 1$ **do**
  $z = (\Phi_i')^T z$
  $z \mathrel{+}= \bar{v}^T \Phi_i'' \dot{v}^{i-1}$
  $\bar{v}^T = \bar{v}^T \Phi_i'$
end
Output: $f''(x)y = Pz$

in its componentwise form. Due to the definition of v^i and (3.14), after the ith iteration of the forward sweep in Algorithm 4.6 we have

$$\dot{v}^i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \nabla v_i^T y \\ \vdots \\ \nabla v_{1-n}^T y \end{bmatrix}. \tag{4.27}$$

Using (4.27) we find that:

$$\bar{v}^T \Phi_i'' \dot{v}^{i-1} = \bar{v}_i [\Phi_i]_i'' \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \nabla v_{i-1}^T y \\ \vdots \\ \nabla v_{1-n}^T y \end{bmatrix} = \bar{v}_i \sum_{j,k} \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \nabla v_k^T \, y.$$

Thus, by storing the gradient-vector products $\nabla v_k^T \, y$ in the kth element of a vector $\dot{v} \in \mathbb{R}^{\ell+n}$ and noting that the pre-multiplication of a vector by $(\Phi_i')^T$ is equivalent to an iteration of the Block format of the Reverse Gradient Algorithm 3.6, Algorithm 4.6 can be written in its componentwise form Algorithm 4.7.

If one restricts the set of elemental functions to binary or unary functions, the complexity of Algorithm 4.7 becomes apparent. Since there are only scalar operations, each iteration of the forward and reverse loop has a complexity of $O(1)$, resulting in a total complexity of $O(\ell)$. This algorithm is used together with a compacting scheme methods [14] to calculate the entire Hessian matrix with only a few Hessian-vector products. The advantageous time

Algorithm 4.7: Reverse Hessian-vector

Input: $x \in \mathbb{R}^n$
Initialization $\bar{v} = e_\ell$, $z = 0 \in \mathbb{R}^{(n+\ell)}$, $\dot{v} = P^T y$
for $i = 1, \ldots, \ell$ **do**
  foreach $j \prec i$ **do** $\dot{v}_i \mathrel{+}= \dfrac{\partial \phi_i}{\partial v_j} \dot{v}_j$
end
for $i = \ell, \ldots, 1$ **do**
  foreach $j \prec i$ **do** $z_j \mathrel{+}= \dfrac{\partial \phi_k}{\partial v_j} z_k$
  foreach $j, k \prec i$ **do**
    $z \mathrel{+}= \bar{v}_i \dfrac{\partial^2 \phi_i}{\partial v_j \partial v_k} \dot{v}_k$
  end
  foreach $j \prec i$ **do** $\bar{v}_j \mathrel{+}= \dfrac{\partial \phi_k}{\partial v_j} \bar{v}_k$
end
Output: $f''(x)y = Pz$

complexity of the Hessian-vector product contributes to make the method relatively efficient, and thus will be used for comparative tests later on.

4.3 A New Hessian Algorithm: `edge_pushing`

4.3.1 Development

In order to arrive at an algorithm to efficiently compute expression (4.17), once again, it is helpful to think in terms of block operations. First of all, rewrite (4.17) as

$$f'' = PWP^T = P \left(\sum_{i=1}^{\ell} W_i \right) P^T, \tag{4.28}$$

so the problem boils down to the computation of W. The summands in W share a common structure, spelled out below for the i-th summand.

$$W_i = \underbrace{\left((\Phi_1')^T \cdots (\Phi_{i-1}')^T \right)}_{\text{left multiplicand}} \quad \underbrace{\left((\bar{\nabla}^i)^T \Phi_i'' \right)}_{\text{central multiplicand}} \quad \underbrace{\left(\Phi_{i-1}' \cdots \Phi_1' \right)}_{\text{right multiplicand}}. \tag{4.29}$$

Using the distributivity of multiplication over addition, the partial sum $W_i + W_{i-1}$ may be expressed as a three multiplicand products where the left and right multiplicands coincide

with those in the expression of W_{i-1}, but the central one is different.

$$W_i + W_{i-1} = \\ \left((\Phi'_1)^T \cdots (\Phi'_{i-2})^T\right)\left((\Phi'_{i-1})^T((\overline{\mathrm{v}}^i)^T\Phi''_i)(\Phi'_{i-1}) + ((\overline{\mathrm{v}}^{i-1})^T\Phi''_{i-1})\right)\left(\Phi'_{i-2}\cdots\Phi'_1\right). \quad (4.30)$$

Instead of calculating each W_i separately, we may save effort by applying this idea to increasing sets of partial sums, all the way to W_ℓ. This alternative way of calculating W is reminiscent of the nested multiplication for polynomials. The nested expression for $\ell = 3$ is given in (4.31) below.

$$W = (\Phi'_1)^T\left((\Phi'_2)^T((\overline{\mathrm{v}}^3)^T\Phi''_3)\Phi'_2 + (\overline{\mathrm{v}}^2)^T\Phi''_2\right)\Phi'_1 + (\overline{\mathrm{v}}^1)^T\Phi''_1. \quad (4.31)$$

Of course, the calculation of such a nested expression must begin at the innermost expression and proceed outwards. This means, in this case, going from the highest to the lowest index. This is naturally accomplished in a backward sweep of the computational graph, which could be schematically described as follows.

$$\begin{array}{rl}
\text{Node } \ell & W \leftarrow (\overline{\mathrm{v}}^\ell)^T\Phi''_\ell \\
\text{Node } \ell-1 & W \leftarrow (\Phi'_{\ell-1})^T W \Phi'_{\ell-1} \\
 & W \leftarrow W + (\overline{\mathrm{v}}^{\ell-1})^T\Phi''_{\ell-1} \\
 & \vdots \\
\text{Node } i & W \leftarrow (\Phi'_i)^T W \Phi'_i \\
 & W \leftarrow W + (\overline{\mathrm{v}}^i)^T\Phi''_i \\
 & \vdots \\
\text{Node } 1 & W \leftarrow (\Phi'_1)^T W \Phi'_1 \\
 & W \leftarrow W + (\overline{\mathrm{v}}^1)^T\Phi''_1.
\end{array}$$

In particular, node ℓ's iteration may be cast in the same format as the other ones if we initialize W as a null matrix.

It follows that the value of W at the end of the iteration where node i is swept is given by

$$W = \sum_{k=i}^{\ell}\left(\prod_{j=i}^{k-1}(\Phi'_j)^T\right)((\overline{\mathrm{v}}^k)^T\Phi''_k)\left(\prod_{j=1}^{k-i}\Phi'_{k-j}\right).$$

Notice that, at the iteration where node i is swept, both assignments involve derivatives of Φ_i, which are available. The other piece of information needed is the vector $\overline{\mathrm{v}}^i$, which we know how to calculate via a backward sweep from Algorithm 3.6. Putting these two together, we arrive at Algorithm 4.8.

Before delving into the componentwise version of Algorithm 4.8, there is a key observation to be made about matrix W, established in the following proposition.

Proposition 4.3 *At the end of the iteration at which node i is swept in Algorithm 4.8, for all i, the nonnull elements of W lie in the upper diagonal block of size $n + i - 1$.*

Algorithm 4.8: Block form of `edge_pushing`.

Input: $x \in \mathbb{R}^n$
initialization: $\overline{\mathrm{v}} = e_\ell$, $W = 0$
for $i = \ell, \ldots, 1$ **do**
  $W = (\Phi_i')^T W \Phi_i'$
  $W += \overline{\mathrm{v}}^T \Phi_i''$
  $\overline{\mathrm{v}}^T = \overline{\mathrm{v}}^T \Phi_i'$
end
Output: $f'' = PWP^T$

Consider the first iteration, at which node ℓ is swept. At the beginning W is null, so the first block assignment $((\Phi_\ell')^T W \Phi_\ell')$ does not change that. Now consider the assignment

$$W \leftarrow W + (\overline{\mathrm{v}})^T \Phi_\ell''.$$

Using (4.10) and the initialization of $\overline{\mathrm{v}}$, we have

$$(\overline{\mathrm{v}})^T \Phi_\ell'' = \bar{v}_\ell [\Phi_\ell]_\ell'' = [\Phi_\ell]_\ell'',$$

and, since $[\Phi_\ell]_\ell(y) = \phi_\ell(y_j)_{j \prec \ell}$, the nonnull entries of $[\Phi_\ell]_\ell''$ must have column and row indices that correspond to predecessors of node ℓ. This means the last row and column, of index ℓ, are zero. Thus the statement of the proposition holds after the first iteration.

Suppose by induction that, after node $i+1$ is swept, the last $\ell - i$ rows and columns of W are null. Recalling (3.10) and using the induction hypothesis, the matrix-product $(\Phi_i')^T W \Phi_i'$ can be written in block form as follows:

$$\left[\begin{array}{c|c|c} I & c^i & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & I \end{array}\right] \left[\begin{array}{c|c|c} W_{1-n..i-1,1-n..i-1} & W_{1-n..i-1,i} & 0 \\ \hline W_{i,1-n..i-1} & w_{ii} & 0 \\ \hline 0 & 0 & 0 \end{array}\right] \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline (c^i)^T & 0 & 0 \\ \hline 0 & 0 & I \end{array}\right] \begin{array}{l} 1-n \\ \vdots \\ i-1 \\ \text{row } i, \\ i+1 \\ \vdots \\ \ell \end{array}$$

which results in

$$\left[\begin{array}{c|c|c} W_{1-n..i-1,1-n..i-1} + c^i W_{i,1-n..i-1} + W_{1-n..i-1,i}\,(c^i)^T + w_{ii}\, c^i (c^i)^T & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \end{array}\right] \begin{array}{l} 1-n \\ \vdots \\ i-1 \\ \text{row } i. \\ i+1 \\ \vdots \\ \ell \end{array} \tag{4.32}$$

Thus at this point the last $\ell - (i-1)$ rows and columns have been zeroed.

Again using (4.10), we have

$$(\overline{\mathrm{v}})^T \Phi_i'' = \bar{v}_i \left(\frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \right)_{1-n \le j,k \le \ell},$$

where the nonnull entries of the Hessian matrix on the right-hand side have column and row indices that correspond to predecessors of node i. Therefore, the last $\ell - (i-1)$ rows and columns of this Hessian are also null. Hence the second and last block assignment involving W will preserve this property, which, by induction, is valid till the end of the algorithm. ∎

Using the definition of c^i in (3.11), the componentwise translation in the first block assignment involving W in Algorithm 4.8 is

$$\left((\Phi_i')^T W \Phi_i'\right)_{jk} = \begin{cases} w_{jk} + \dfrac{\partial \phi_i}{\partial v_k}\dfrac{\partial \phi_i}{\partial v_j} w_{ii} + \dfrac{\partial \phi_i}{\partial v_k} w_{ji} + \dfrac{\partial \phi_i}{\partial v_j} w_{ik}, & \text{if } j < i \text{ and } k < i, \\ 0, & \text{otherwise.} \end{cases} \tag{4.33}$$

For the second block assignment, using (4.10), we have that

$$\left((\overline{\mathrm{v}})^T \Phi_i''\right)_{jk} = \begin{cases} \bar{v}_i \dfrac{\partial^2 \phi_i}{\partial v_j \partial v_k}, & \text{if } j < i \text{ and } k < i, \\ 0, & \text{otherwise.} \end{cases} \tag{4.34}$$

Finally, notice that, since the componentwise version of the block assignment, done as node i is swept, involves only entries with row and column indices smaller than or equal to i, one does not need to actually zero out the row and column i of W, as these entries will not be used in the following iterations.

This componentwise assignment may be still simplified using symmetry, since W's symmetry is preserved throughout `edge_pushing`. In order to avoid unnecessary calculations with symmetric counterparts, we employ the notation $w_{\{ji\}}$ to denote both w_{ij} and w_{ji}. Notice, however, that, when $j = k$ in (4.33), we have

$$\left((\Phi_i')^T W \Phi_i'\right)_{jj} = w_{jj} + \left(\frac{\partial \phi_i}{\partial v_j}\right)^2 w_{ii} + \frac{\partial \phi_i}{\partial v_j} w_{ji} + \frac{\partial \phi_i}{\partial v_j} w_{ij},$$

so in the new notation we would have

$$\left((\Phi_i')^T W \Phi_i'\right)_{\{jj\}} = w_{\{jj\}} + \left(\frac{\partial \phi_i}{\partial v_j}\right)^2 w_{\{ii\}} + 2\frac{\partial \phi_i}{\partial v_j} w_{\{ji\}}.$$

The componentwise version of Algorithm 3 adopts the point of view of the node being swept. Say, for instance that node i is being swept. Consider the first block assignment

$$W \leftarrow (\Phi_i')^T W \Phi_i',$$

whose componentwise version is given in (4.33). Instead of focusing on updating each $w_{\{jk\}}$, $j, k < i$, at once, which would involve accessing $w_{\{ii\}}$, $w_{\{ji\}}$ and $w_{\{ik\}}$, we focus on each

$w_{\{pi\}}$ at a time, and 'push' its contribution to the appropriate $w_{\{jk\}}$'s. Taking into account that the partial derivatives of ϕ_i may only be nonnull with respect to i's predecessors, these appropriate elements will be $w_{\{jp\}}$, where $j \prec i$, see the `pushing` step in Algorithm 4.9.

The second block assignment

$$W \leftarrow W + (\overline{\mathrm{v}}^i)^T \Phi_i'',$$

may be thought of as the creation of new contributions, that are added to appropriate entries and that will be pushed in later iterations. From its componentwise version in (4.34), we see that only entries of W associated with predecessors of node i may be changed in this step. The resulting componentwise version of the `edge_pushing` algorithm is Algorithm 4.9.

Algorithm 4.9 has a very natural interpretation in terms of the graph model introduced in Section 4.1.2. The nonlinear arcs are 'created' and their weight initialized (or updated, if in fact they already exist) in the `creating` step. In graph terms, the `pushing` step performed when node i is swept actually pushes the endpoints of the nonlinear arcs incident to node i to its predecessors. The idea is that subpaths containing the nonlinear arc are replaced by shortcuts. This follows from the fact that if a path contains the nonlinear arc $\{i, p\}$, then it must also contain precisely one of the other arcs incident to node i. Figure 4.4 illustrates the possible subpaths and corresponding shortcuts. In cases I and III, the subpaths consist of two arcs, whereas in case II, three arcs are replaced by a new nonlinear arc. Notice that the endpoints of a loop (case II) may be pushed together down the same node, or split down different nodes. In this way, the contribution of each nonlinear arcs trickles down the graph, distancing the higher numbered nodes until it finally reaches the independent nodes.

This interpretation helps in understanding the good performance of `edge_pushing` in the computational tests, in the sense that only "proven" contributions to the Hessian (nonlinear arcs) are dealt with.

What differentiates this `edge_pushing` from the simple presentation in Algorithm 4.1, is that now the execution can be carried out in the two orderly sweeps of the intermediate variables and we have adapted to using undirected nonlinear edges. But more considerations have to be made to implement this code, for we have yet to make suppositions about our set of elemental functions, consider possible overwrites and how to take more advantage of sparsity. These details and others will be discussed in Chapter 5.

4.3.2 Examples

In this section we run Algorithm 4.9 on two examples, to better illustrate its workings. Since we're doing it on paper, we have the luxury of doing it symbolically. In the first example we show all the workings of each iteration of `edge_pushing`, in the second we show an overview on a larger example.

Example 1:
The iterations of `edge_pushing` on a computational graph of the function $f(x) = (x_{-2} +$

Algorithm 4.9: Componentwise form of `edge_pushing`.

Input: $x \in \mathbb{R}^n$

initialization: $\bar{v}_{1-n} = \cdots = \bar{v}_{\ell-1} = 0$, $\bar{v}_\ell = 1$, $w_{\{ij\}} = 0$, $1 - n \le j \le i \le \ell$

for $i = \ell, \ldots, 1$ **do**

`Pushing`

foreach *p such that* $p \le i$ *and* $w_{\{pi\}} \neq 0$ **do**

if $p \neq i$ **then**

foreach $j \prec i$ **do**

if $j = p$ **then**

$$w_{\{pp\}} += 2\frac{\partial \phi_i}{\partial v_p} w_{\{pi\}}$$

else

$$w_{\{jp\}} += \frac{\partial \phi_i}{\partial v_j} w_{\{pi\}}$$

end

end

else $p = i$

foreach *unordered pair* $\{j, k\}$ *such that* $j, k \prec i$ **do**

$$w_{\{jk\}} += \frac{\partial \phi_i}{\partial v_k}\frac{\partial \phi_i}{\partial v_j} w_{\{ii\}}$$

end

end

end

`Creating`

foreach *unordered pair* $\{j, k\}$ *such that* $j, k \prec i$ **do**

$$w_{\{jk\}} += \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_k \partial v_j}$$

end

`Adjoint`

foreach $j \prec i$ **do**

$$\bar{v}_j += \bar{v}_i \frac{\partial \phi_i}{\partial v_j}$$

end

end

Output: $f'' = PWP^T$

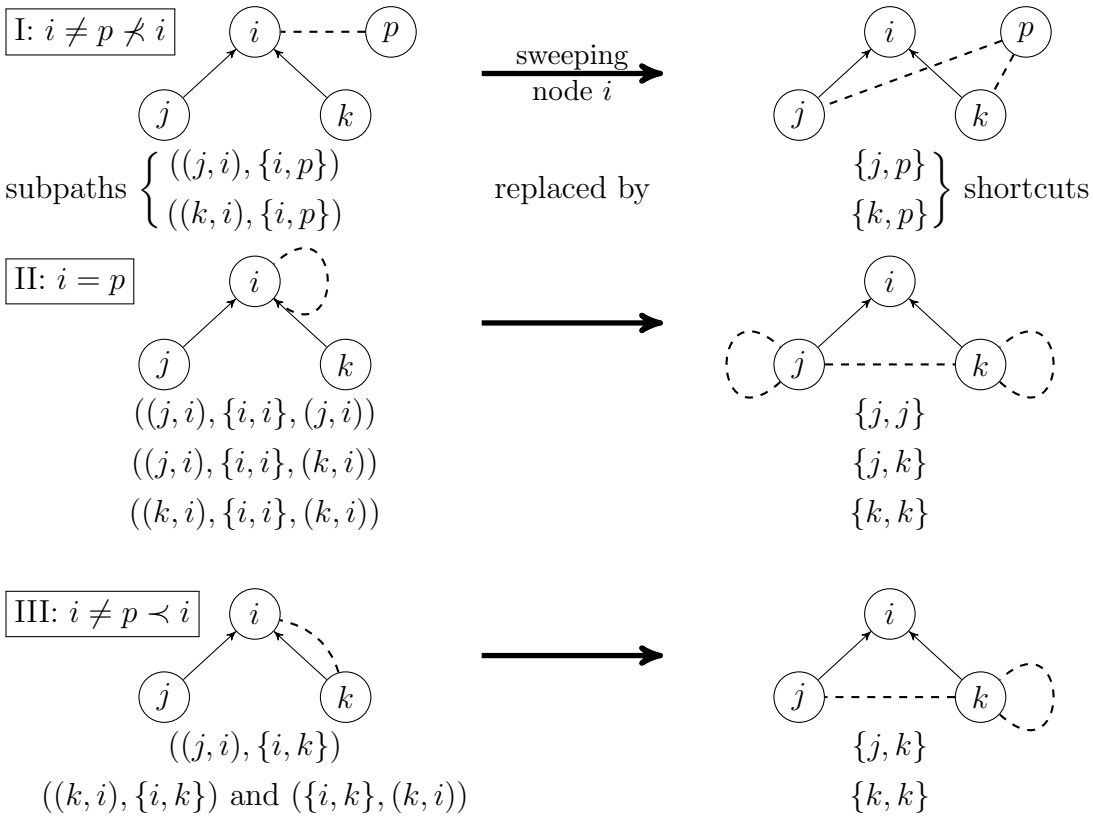


Figure 4.4: Pushing nonlinear arc $\{i,p\}$ is creating shortcuts.

$e^{x_{-1}})(3x_{-1} + x_0^2)$ are shown on Figure 4.5. The thick arrows indicate the sequence of three iterations. Nodes about to be swept are highlighted. As we proceed to the graph on the right of the arrow, nonlinear arcs are created (or updated), weights are appended to edges and adjoint values are updated, except for the independent nodes, since the focus is not gradient computation. For instance, when node 3 is swept, the nonlinear arc $\{1, 2\}$ is created. This nonlinear arc is pushed and split into two when node 2 is swept, becoming nonlinear arcs $\{0, 1\}$ and $\{-1, 1\}$, with weights $1 \cdot 2v_0$ and $1 \cdot 3$, respectively. When node 1 is swept, the nonlinear arc $\{-1, 1\}$ is pushed and split into nonlinear arcs $\{-2, -1\}$ and $\{-1, -1\}$, the latter with weight $2 \cdot 3 \cdot e^{v_{-1}}$. Later on, in the same iteration, the nonlinear contribution of node 1, $\partial^2\phi_1/\partial v_{-1}^2$, is added to the nonlinear arc $\{-1, -1\}$. Other operations are analogous. The Hessian can be retrieved from the weights of the nonlinear arcs between independent nodes at the end of the algorithm:

$$f''(x) = \begin{pmatrix} 0 & 3 & 2v_0 \\ 3 & e^{v_{-1}}(6 + v_2) & 2v_0 e^{v_{-1}} \\ 2v_0 & 2v_0 e^{v_{-1}} & 2v_1 \end{pmatrix} = \begin{pmatrix} 0 & 3 & 2x_0 \\ 3 & e^{x_{-1}}(6 + 3x_{-1} + x_0^2) & 2x_0 e^{x_{-1}} \\ 2x_0 & 2x_0 e^{x_{-1}} & 2(x_{-2} + e^{x_{-1}}) \end{pmatrix}.$$

Notice that arcs that are pushed are deleted from the figure just for clarity purposes, though this is not explicitly done in Algorithm 4.9. Nevertheless, in the actual implementation the memory locations corresponding to these arcs are indeed deleted, or in other words, made available for overwritting.

Example 2:
In Figure 4.6 we depict the execution of `edge_pushing` on the function

$$f(x_{-2}, x_{-1}, x_0) = (x_{-2} + 1)(x_{-1} + 1)3(x_0 + 1).$$

The evaluation on this function is broken up into a list of intermediate variables which are in a table on the top lefthand side of Figure 4.6. In this same table there are also a few adjoint values which are needed in the execution, and we omit the details on how the adjoints calculated. The thick arrows indicate the sequence of six iterations. Nodes about to be swept are highlighted. As we proceed to the graph on the right of the arrow, nonlinear arcs are created (or updated), weights are appended to edges. For instance, when node 4 is swept, the nonlinear arc $\{3, 4\}$ is pushed to its predecessores and split into two edges: $\{1, 3\}$ and $\{2, 3\}$ with weights $3v_2$ and $3v_1$, respectively. The edge $\{1, 2\}$ is then created with weight v_5, for $\phi_4 = v_1 v_2$ is a nonlinear function.

No parallel reductions occur during the execution of this function, thus the weight of each nonlinear arc is determined during its allocation and is not altered in future iterations. Drawing the computational graph with independent nodes lower down and the output node at the top, `edge_pushing` creates edges at higher levels of the graph that trickle down to the independent nodes.

The Hessian can be retrieved from the weights of the nonlinear arcs between independent

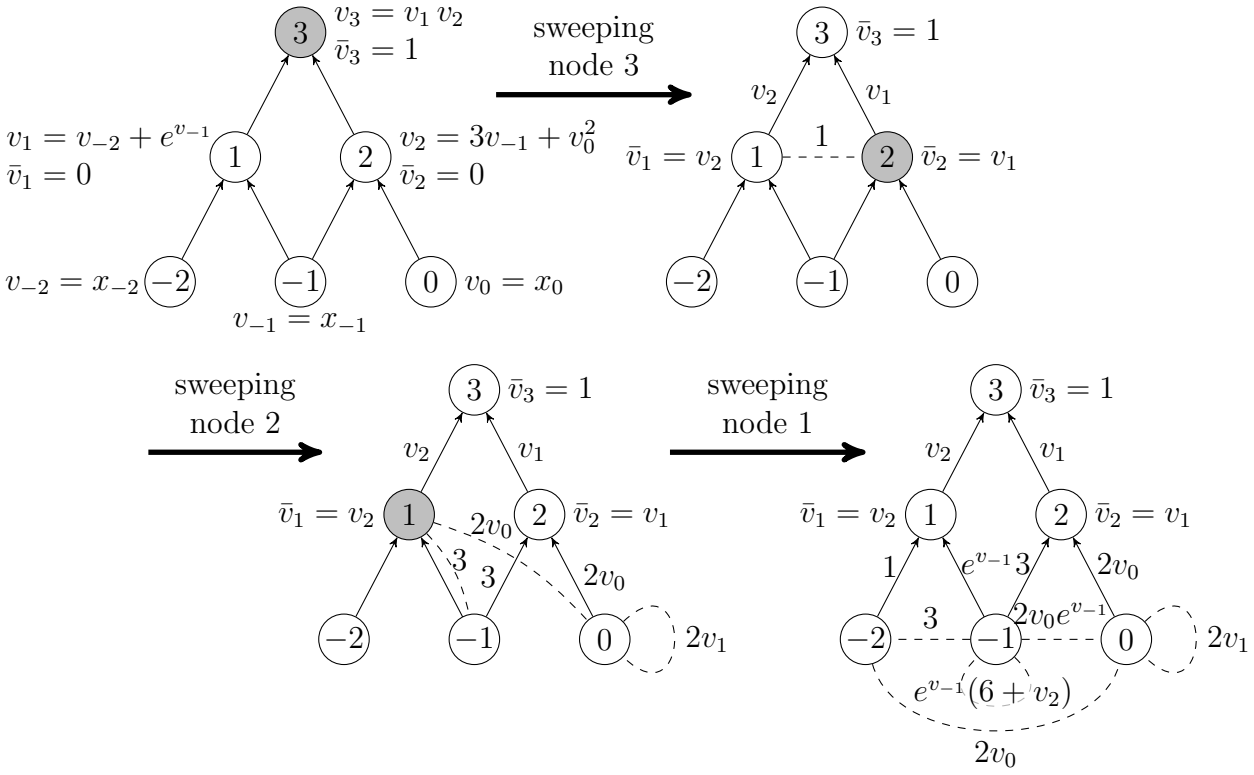


Figure 4.5: `edge_pushing` applied to a computational graph of $f(x) = (x_{-2}+e^{x_{-1}})(3x_{-1}+x_0^2)$.

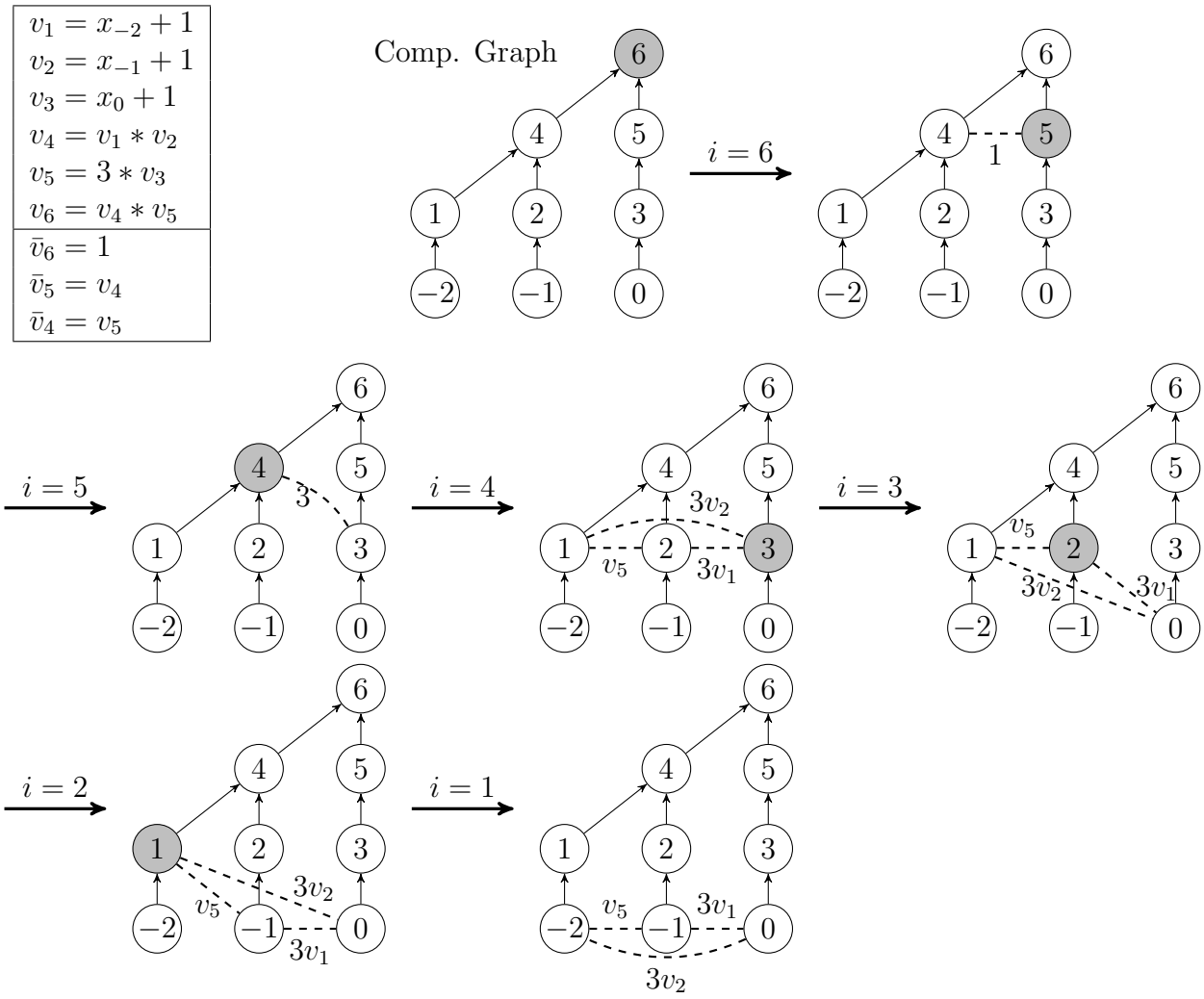


Figure 4.6: $f(x_{-2}, x_{-1}, x_0) = (x_{-2} + 1)(x_{-1} + 1)3(x_0 + 1)$

nodes at the end of the algorithm:

$$f'' = \begin{pmatrix} 0 & v_5 & 3v_2 \\ v_5 & 0 & 3v_1 \\ 3v_2 & 3v_1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 3(x_0+1) & 3(x_{-1}+1) \\ 3(x_0+1) & 0 & 3(x_{-2}+1) \\ 3(x_{-1}+1) & 3(x_{-2}+1) & 0 \end{pmatrix}.$$

Chapter 5

Implementing the New Reverse Hessian Algorithm

5.1 Implementation Issues of Reverse Modes

A major challenge in designing reverse automatic differentiation procedures is *running the evaluation procedure backwards.* To this end, one usually records information during the function evaluation on what is called a *Tape* $\mathcal{T}$. The Tape is essentially a first-in-last-out data structure capable of storing large amounts of sequential access memory. This Tape $\mathcal{T}$ must contain the necessary information to represent the computational graph of the function and allow the calculation of first and second order derivatives of the intermediate variables during a reverse sweep. A possible solution is, for each intermediate variable v_i, to record on $\mathcal{T}$ the value attributed to v_i, a number identifying the elemental function ϕ_i and the indices of the input variables, e.g., if $v_i = v_k v_j$, we record v_i, a number that identifies the multiplication function, and the indices k and j. This information enables us to calculate all of ϕ_i's partial derivatives. Such a Tape can be generated automatically during the evaluation of a function program using operator overloading.

Operator overloading is a tool in many modern programming languages such as C++ and FORTRAN 90, which allows the programmer to give alternative implementations for the same operator when called with different data types. For instance, defining a new type of variable called **adouble**, we can implement a function sin(**adouble** x) without altering the standard definition for sin(**double** x). This is very convenient, for we can program our new function sin(**adouble** x) to not only calculate sin(**double** x), but also to record a new entry on a Tape $\mathcal{T}$. For instance, sin(**adouble** x) could be programmed to perform three functions: calculate the floating point value of $\sin(x)$, record on $\mathcal{T}$ the index of x and an integer `sinv` that is a reference to the function $\sin(\cdot)$.

If such an operator overloading strategy were implemented, then the user would only have to change the data type used in the function program to **adouble**, as done in Algorithm 5.1, and upon evaluation, a Tape would be automatically generated. The quantity in bytes occupied by an element recorded on this Tape is bounded by a constant, hence the total

Algorithm 5.1: A Function Program

Input: **adouble** (x_1, x_2, x_3)

Initialization: **adouble** $f = 0$

$$f = \cos(x_1) * \left(\frac{3x_2}{x_1}\right) + 3 * x_2 * \exp(x_3)$$

Output: f

of Sequential Access Memory (SAM) used is proportional to ℓ: the number of intermediate variables. This is potentially a lot of memory, so it is possible that this Tape has to be recorded on the hard disk instead of the internal memory of the processor. Though one can reduce the quantity of information recorded by taking into consideration that some functions can be recalculated on the way back, while others have partial derivatives that are independent of the input variables, such as linear and constant functions.

Better bounds on SAM usage are attained by a method by Griewank and Walther [18, 31] that involves saving snapshots of the evaluation procedure and recalculating from snapshot to snapshot. This method, called check-pointing, requires a quantity proportional to $\log(\ell)$ of memory. This is an important result for reverse procedures, for a usage of SAM proportional to ℓ made reverse procedures prohibitive for some very large scale problems.

Reverse modes also require a portion of Random Access Memory (RAM). For instance, as node i is swept in the Reverse Gradient Algorithm 3.8, one needs instant access to the adjoints of i's predecessors. Though there are ℓ distinct adjoints, one may use significantly less if an overwriting scheme is employed. Note how in the code for the Black-Box Gradient in Algorithm 3.1 the internal variables a and b are reused or, in other words, their previous content is overwritten by new values. With the exception of the Black-Box Gradient Algorithm 3.1, all the AD algorithms presented so far employ no explicit overwriting. Take for example the Reverse Gradient Algorithm 3.8, the ith adjoint value is stored in a distinct location designated by $\bar{v}_i$, but in an efficient implementation, the adjoints will be stored in internal variables that are reused.

Let us now address the problem of how much RAM is required to execute the Reverse Gradient Algorithm. To bound how many distinct internal variables are required, we must take into account the the *lifespan* of a variable, where the lifespan of a data element used in an algorithm is the maximum interval of iterations in which we require access to it.

As node i is being swept in Algorithm 3.8, to evaluate the partial derivatives of ϕ_i we need access to the values of the arguments v_j, $j \prec i$ and to calculate adjoint values we need access to $\bar{v}_i$ and $\bar{v}_j$, $j \prec i$. There is no predefined order in which we will access these floating point values, hence it is preferable to store these values on a structure that permits random access, being a vector structure the usual choice. For $\overline{v}_i$ its lifespan starts at the first incrementation,

$$\overline{v}_i += \overline{v}_k \frac{\partial \phi_k}{\partial v_i},$$

where $k = \max\{s \,|\, i \prec s\}$, and ends after the last use in an argument on the right hand side,

$$\overline{v}_j += \overline{v}_i \frac{\partial \phi_i}{\partial v_j},$$

where $j = \min\{s \,|\, s \prec i\}$.

The lifespan of v_i in a reverse procedure is necessarily shorter then that of $\overline{v}_i$, starting with its first use in an argument, on iteration k such that $k = \max\{s \,|\, i \prec s\}$ and ending after being used as an argument for the last time on iteration $j = \min\{s \,|\, i \prec s\}$. Let r be the maximum number of adjoint variables that have an overlapping lifespan. We refer to r as the *maximum number of live variables*. A vector of size r for the adjoints and another one for the v_i values would suffice to guarantee random access when needed.

In most applications r tends to be much smaller then ℓ. This allows us to store these vector structures in the internal memory of the processor, which in turn permits fast access. It will be shown later that, with a specific data structure, one can implement the `edge_pushing` Algorithm 4.9 in such a way that only a quantity of RAM proportional to r is necessary.

5.2 The Adjacency List Data Structure

The core data structure used in our implementations is the Adjacency List. It is a structure primarily used to represent graphs. First, additional notation is required. For an undirected graph $G = (V, E)$, the set of neighbors of node i is denoted by $N_i \equiv \{j \mid \exists\{j, i\} \in E\}$. The degree of node i is defined and denoted by $d_i \equiv |N_i|$.

Consider an undirected graph $G = (V, E)$ with $|V| = n$ nodes, the Adjacency List structure is comprised of an array with n positions, and each position contains a pointer to a linked list. Turn to the example of a graph and a corresponding Adjacency List in Figure 5.1. The ith position in the array is associated to the ith node in the graph, and connected to it is an ordered list of its neighbors. Additionally, if a graph has a weight function $w : E \to \mathbb{R}$, in the list connected to position i, we store the weight w_{ij} in node j.

Let us examine the complexity of two basic operations on the data structure: **Find** and **Insertion**, so we may in turn bound the complexity of the algorithms that use such a structure.

- **Find:** To search for an edge $\{i, j\}$ or a weight w_{ij}, we may check either the list connected to position i or j. In searching down the list adjacent to i (resp., j), it is possible that we will go through all d_i (resp., d_j) elements in the list. Hence the time to find edge $\{i, j\}$ or weight w_{ij} is bounded by $O(d_i + d_j)$. The only exception to this is if $i = j$, in which case it is a loop such as the edge $\{1, 1\}$ in Figure 5.1. Finding a loop is bounded by $O(d_i)$.

- **Insertion:** The nodes in each list are arranged in increasing order, which must be preserved when inserting a new edge. The insertion of a new edge $\{i, j\}$ implies changes

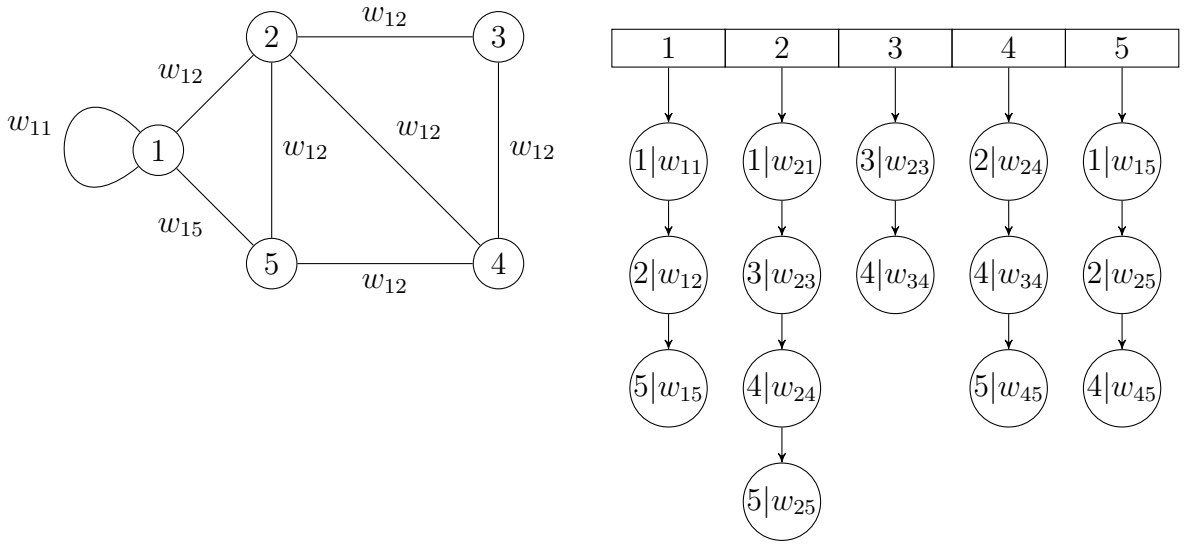


Figure 5.1: A graph and its representation as an Adjacency List

to the lists connected to positions i and j. In the worst case, one must scan the whole list in order to insert an element therein. Thus, the insertion of a new edge $\{i, j\}$ has a complexity of $O(d_i + d_j)$.

5.3 An Implemention of the `edge_pushing` Algorithm

We have implemented the `edge_pushing` Algorithm 4.9 in C++ and integrated the code into the automatic differentiation package ADOL-C [19]. ADOL-C uses operator overloading to generate a Taped evaluation procedure $\mathcal{T}$. This Tape $\mathcal{T}$ and a point $x \in \mathbb{R}^n$ form the input to the `edge_pushing` Algorithm 5.2. The output is a sparse representation of the hessian matrix $f''(x)$.

$$\texttt{edge\_pushing}(\underset{x}{\downarrow}, \underset{\mathcal{T}}{\downarrow}, \underset{f''(x)}{\downarrow})$$

The first step in elaborating an implementation is choosing appropriate data structures. Our main concern is the choice of the structure for the matrix W used in Algorithm 4.9. Our choice must be suitable for symmetric sparse matrices. Sparse, for Hessian matrices tend to be sparse in many applications. For these reasons we chose the Adjacency List structure, section 5.2. Each nonzero $w_{\{jk\}}$ in Algorithm 4.9 will be represented by an edge $\{j, k\}$ and a weight $w_{\{jk\}}$. The data structure stores a graph $G = (V \cup Z, N)$, where $V = \{1, \ldots, \ell\}$, $Z = \{1 - n, \ldots, 0\}$, $N \subset \{\, \{j, k\} \,|\, j, k \in V \cup Z\}$ and a weight function $w : N \to \mathbb{R}$. The pseudocode for our implementation is in Algorithm 5.2. We assume that the set of elemental functions is composed of only unary and binary functions.

For simplicity, we have not made explicit the allocation of the edges $\{j, k\}$. Instead we only refer to the edges weights $w_{\{j,k\}}$. An edge will be allocated as soon as its weight assumes a nonzero value.

Now that we have chosen a data structure we can establish bounds on the time and memory complexity of the algorithm.

5.4 Complexity Bounds

5.4.1 Time complexity

During the execution of the algorithm, new arcs may be allocated in the structure G during the `pushing` or the `creating` step. After node i has been swept, G has accumulated all the nonlinear arcs that have been created or pushed, up to this iteration, since arcs are not deleted. One may think of G as the recorded history (creation and pushing) of the nonlinear arcs.

The time complexity of `edge_pushing` depends on how many nonlinear arcs are allocated during execution. Thus it is important to establish bounds for the number of arcs allocated to each node. Of course the degree of node i and its neighborhood vary during the execution of the algorithm. To this end, we fix G^* as the graph obtained at the end of the algorithm. Recall that edges are not deleted in Algorithm 5.2.

Let d_i^* be the degree of node i in G^*, and let $d^* = \max_i\{d_i^*\}$. Clearly $d_i \leq d_i^*$, where d_i is the degree of node i in the graph G at any given iteration, for no edges are deleted during the

Algorithm 5.2: `edge_pushing`

Input: $x \in \mathbb{R}^n$, a Taped evaluation procedure $\mathcal{T}$ of $f(x)$

Intialization:

$\bar{v} = e_{\ell+n}$

$G = (V \cup Z, N)$ with $|V \cup Z| = n + \ell$, $N = \emptyset$

for $i = \ell, \ldots, 1$ **do**

`Pushing:`

foreach $\{i, p\} \in N$ **do**

if $p \neq i$ **then**

foreach $j \prec i$ **do**

if $j = p$ **then**

$w_{\{p,p\}} \mathrel{+}= 2\frac{\partial \phi_i}{\partial v_p} w_{\{i,p\}}$

else

$w_{\{j,p\}} \mathrel{+}= \frac{\partial \phi_i}{\partial v_j} w_{\{i,p\}}$

end

if $p = i$ **then**

foreach *unordered pair* $\{j, k\}$ *such that* $j, k \prec i$ **do**

$w_{\{j,k\}} \mathrel{+}= \frac{\partial \phi_i}{\partial v_k} \frac{\partial \phi_i}{\partial v_j} w_{\{i,i\}}$

end

end

`Creating:`

if ϕ_i *is a nonlinear function* **then**

if ϕ_i *is unary* $v_i = \phi_i(v_j)$ *and* $\frac{\partial^2 \phi_i}{\partial v_j^2} \neq 0$ **then**

$w_{\{j,j\}} \mathrel{+}= \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_j^2}$

if ϕ_i *is binary* $v_i = \phi_i(v_j, v_k)$ **then**

if $\frac{\partial^2 \phi_i}{\partial v_j^2} \neq 0$ **then**

$w_{\{j,j\}} \mathrel{+}= \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_j^2}$

if $\frac{\partial^2 \phi_i}{\partial v_k \partial v_j} \neq 0$ **then**

$w_{\{k,j\}} \mathrel{+}= \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_k \partial v_j}$

if $\frac{\partial^2 \phi_i}{\partial v_k^2} \neq 0$ **then**

$w_{\{k,k\}} \mathrel{+}= \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_k^2}$

`Adjoint:`

foreach j *such that* $j \prec i$ **do**

$\bar{v}_j \mathrel{+}= \bar{v}_i \frac{\partial \phi_i}{\partial v_j}$

end

end

Output: $\frac{\partial^2 f(x)}{\partial x_j \partial x_k} = w_{\{j-n,k-n\}}$ **foreach** $w_{\{j-n,k-n\}} \neq 0$

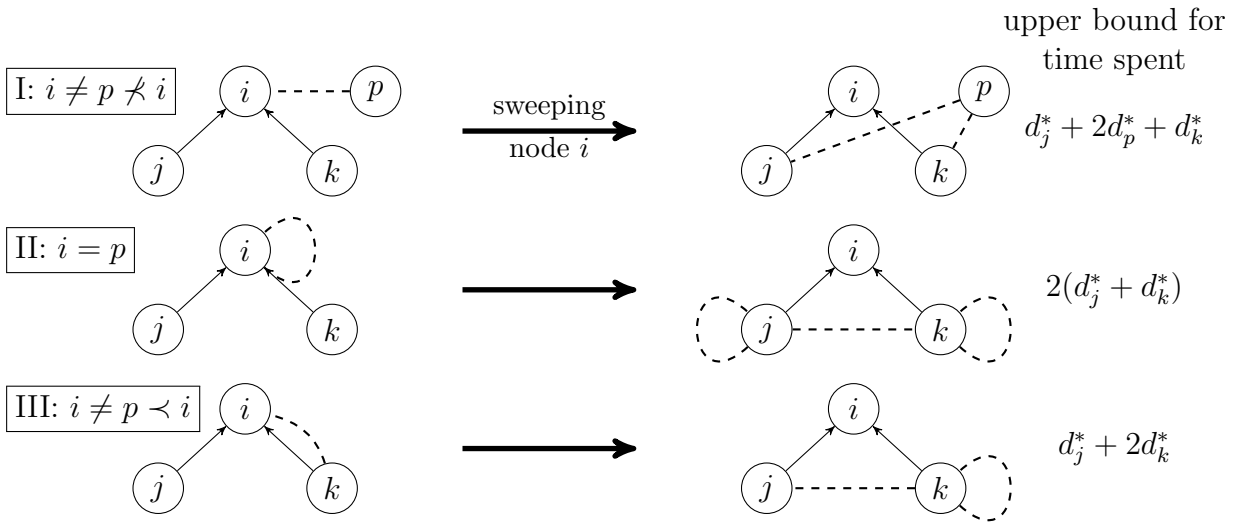


Figure 5.2: Complexity bounds for the pushing step.

execution of Algorithm 5.2. In order to bound the complexity of `edge_pushing`, we consider the `pushing` and `creating` steps separately. We repeat in Figure 5.2 the possible cases of pushing, and the corresponding time complexity bounds. On the right of Figure 5.2 we have the new edges allocated after pushing the corresponding edge on the left.

Studying the cases spelled out in Figure 5.2, one concludes that the time spent in pushing edge $\{i, p\}$ is bounded by $2(d_j^* + d_p^* + d_k^*)$, where j and k are predecessors of node i. Since there are at most d_i^* nonlinear arcs incident to node i, the time spent in the `pushing` step at the iteration where node i is swept is bounded by

$$d_i^*(2(d_j^* + d_p^* + d_k^*)) = O(d_i^*(d_j^* + d_p^* + d_k^*)) = O(d_i^* d^*).$$

Finally, the assumption that all functions are either unary or binary implies that at most three nonlinear arcs are allocated between predecessors during the `creating` step, analogous to case II in Figure 5.2. Hence the time used up in this step at the iteration where node i is swept is bounded by

$$2(d_j^* + d_k^*) = O(d_j^* + d_k^*) = O(d^*),$$

where j and k are predecessors of node i.

Thus, taking into account the time spent in merely visiting a node — say, when the intermediate function associated with the node is linear — is constant, the time complexity

of `edge_pushing` is

$$\begin{aligned} \text{TIME}(\texttt{edge\_pushing}) &\leq \sum_{i=1}^{\ell}(d_i^* d^* + d^* + 1) \\ &= O\left(d^* \sum_{i=1}^{\ell} d_i^* + \ell\right). \end{aligned} \tag{5.1}$$

A consequence of this bound is that, if f is linear, the complexity of `edge_pushing` is that of the function evaluation, a desirable property for Hessian algorithms. This bound is possibly too pessimistic in general, but can be used to provide more meaningful estimates for the class of partially separable functions (which are going to be defined in a later section). Unfortunately, the bounds do not employ the "natural" parameters of the input (n, ℓ), so as to render it not very useful to end-users.

5.4.2 Memory Complexity Bound

To save space, we allow overwrites of edges. Bounds on the usage of memory of the `edge_pushing` Algorithm 5.2 are summed up in Proposition 5.1.

Proposition 5.1 *The* ***`edge_pushing`*** *Algorithm 5.2 uses*

(i) $O(r)$ *of RAM*

(ii) $O(rd^*)$ *of SAM.*

Here r is the maximum number of lives variables. To prove Proposition 5.1 we must consider the lifespan of an edge, as defined on page 46.

Proposition 5.2 *If an edge* $\{j, k\}$ *is allocated, then the adjoints* $\bar{v}_i$ *and* $\bar{v}_j$ *are alive.*

Demonstration: First we prove the assertation for any edge $\{j, k\}$ allocated during the `Creating` step, thus j and k are predecessors of a node i that is being swept. The next subprocedure after the `Creating` step in the algorithm is the `Adjoint` step, where the adjoint of all predecessors are incremented, including $\bar{v}_j$ and $\bar{v}_k$. So the assertion holds true for edges allocated in the `Creating` step.

We will prove by induction that the assertion is true at the end of each iteration. Since N is initially empty, edges are only allocated on the first iteration through the `Creating` step, thus the assertion holds true at its end.

Suppose by induction that the assertion holds at the end of the iteration at which node $m+1$ is swept. Consider the sweeping of node m. Let $\{j, k\}$ be an edge allocated during the `Pushing` step, then without loss of generality, there existed an edge $\{m, p\}$ such that $j \prec m$ and ($k \prec m$ or $k = p$). If k (resp., j) is a predecessor of m, then the adjoint $\bar{v}_k$ (resp., $\bar{v}_j$) must be alive for the `Adjoint` step of iteration m. Else if $k = p$, then due to the induction

hypothesis $\bar{v}_p$ was alive when $\{m, p\}$ was allocated and will continue alive until node p is visited. Hence $\bar{v}_p$ was alive when $\{j, k\}$ was allocated. Therefore, the assertion is true at the end of the sweeping of node m, and, by induction, at the end of all iterations. ∎

Proposition 5.3 *The lifespan of an edge $\{j, k\}$ is contained in the intersection of the lifespans of the adjoints $\bar{v}_j$ and $\bar{v}_k$.*

Demonstration: In Proposition 5.2 we proved that when $\{j, k\}$ allocated, the corresponding adjoints were alive. The edge $\{j, k\}$ will be deleted when pushed on iteration $\max\{j, k\}$ until which the adjoints $\bar{v}_j$ and $\bar{v}_k$ will be alive, for when the node j (resp., k) is swept, $\bar{v}_j$ (resp., $\bar{v}_k$) is used to calculate the adjoints of predecessores. ∎

Demonstration of Proposition 5.1 : The first item is a consequence of Proposition 5.3, for we only need r distinct locations for the nodes in our graph structure. To prove this, supppose that on one iteration there are more then r nodes with adjacent edges. As a consequence of Proposition 5.3 there are more then r adjoints simultaneaosly alive, which is impossible for r is the maximum number of adjoints that have an overlapping lifespan. The Adjacency List structure uses a quantity of RAM proportional to the number of nodes. This proves the first item of Proposition 5.1. The second item is a consequences of the first item, for if there are r nodes in the graph, and each node has at most d^* neighbors then there are at most rd^* elements in the adjacency lists and lists are a form of SAM. ∎

The `edge_pushing` Algorithm uses as much RAM as the Reverse Gradient Algorithm 3.8. The question is if the quantity of SAM used is too much to keep on the internal memory of the processor. Though we do not report on all such tests, we carried out experiments on functions with very sparse Hessians with more then 10^6 variables, and no overflow of the internal memory occurred.

5.4.3 Bounds for Partially Separable Functions

As defined in [27], a function $f : \mathbb{R}^n \to \mathbb{R}$ is partially separable if

$$f(x) = \sum_{i=1}^{m} f_i(x_{I_i}), \tag{5.2}$$

where $I_i \subset \{1, \ldots, n\}$ and $\cup_{i=1}^{m} I_i = \{1, \ldots, n\}$, and there exists $p \in \mathbb{N}$ such that: $|I_i| \leq p < n$ for $i = 1, \ldots, m$.

Additionally, let us assume we have a coded instance of a partially separable function and let q be an integer such that the number of intermediate variables used to calculate $f_i(x_{I_i})$ is bounded above by q, for $i = 1, \ldots, m$. When $p \ll n$, this representation can be exploited in a number of contexts such as nonlinear optimization [27] and efficiently calculating derivatives [21].

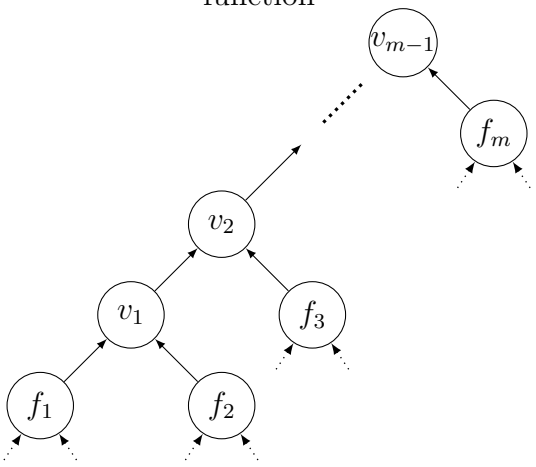
The computational graph in Figure 5.3 is compatible with the evaluation of a partially separable function. Each node f_i is the root of a computational graph that calculates the nonlinear function $f_i(x_{I_i})$. The drawing does not indicate such, but the f_i subgraphs may

share nodes and independent variables with each other. The emphasis in the schematics is how these graphs are connected from above, leaving possible connections from below to the imagination. The v_j nodes are the partial sums:

$$v_j = \sum_{i=1}^{j} f_i(x_{I_i}) \tag{5.3}$$

When using operator overloading to generate an evaluation procedure, the format of the computational graph in Figure 5.3 is very common for partially separable functions. What we set out to demonstrate here does not require that the computional graph be exacly like Figure 5.3, but simply that the last $m-1$ intermediate variables of the evaluation procedure are sums of the f_i's. The computational graph in Figure 5.3 is an example of such an evaluation procedure, and will serve as a visual guide for our demonstration. Upon executing the `edge_pushing` Algorithm on such a evaluation procedure, the first $m-1$ nodes visited are sum operations hence no edges will be allocated. Hence most all of the computational effort occurs while the nodes in the f_i rooted graphs are swept. For functions such that $p \ll m$ and $q \ll m$, the `edge_pushing` algorithm efficiently calculates the Hessian matrix, Proposition 5.4.

Figure 5.3: Computational graph associated to partially separable function



Proposition 5.4 *Consider the partially separable function in (5.2). Let p and q be an upperbound for the number independent and intermediate variables, respectively, in each f_i subgraph, for all $i \in \{1, \ldots, m\}$. Assume a computational graph $G = (V, E)$ for f such that the last $m-1$ node of G are the partial sum variables (5.3), then the* ***edge_pushing*** *Algorithm 4.9 has a complexity of $O(mp^2q^3)$.*

Demonstration: The number of edges incident to any node within the computational graph rooted by f_i, is bounded by the number of nodes therein, ergo $d^* \leq (q+p)$. Applying the bound in (5.1) to a single f_i subgraph, the complexity is bounded by:

$$O\left(\sum_{i=1}^{q}(q+p)^2\right) = O\left(q(q+p)^2\right).$$

Hence the total complexity adds up to the computation of the m graphs rooted by f_i, $i = 1, \ldots, m$, and visiting all the nodes v_j, $j = 1, \ldots, m-1$:

$$O\left(m(q(q+p)^2) + m - 1\right) = O\left(mp^2q^3\right). \quad \blacksquare$$

We have found throughout our numerical experiments that for partially separable functions that are re-scalable in dimension (n), the number of nonlinear terms m will have a linear relation with n while p and q tend to be fixed and independent of dimension. We will emphasize this linear dependency by plotting the time taken by `edge_pushing` over varying dimensions with examples from the CUTE collection.

A fundamental aspect of the `edge_pushing` Algorithm 4.9 is that the calculations carried out on each nonlinear term of a partially separable function, are disjoint from the calculations on other nonlinear terms. Therefore the execution on a partially separable function of m terms can be broken down into m disjoint set of calculations, hence the workload can be spread across m processors.

5.5 Computational experiments

All tests were run on the 32-bit operating system Ubuntu 9.10, processor Intel 2.8 GHz, and 4 GB of RAM. All algorithms were coded in C and C++. The algorithm `edge_pushing` has been implemented as a driver of ADOL-C, and uses the taping and operator overloading functions of ADOL-C [19]. The tests aim to establish a comparison between `edge_pushing` and two algorithms, available as drivers of ADOL-C v. 2.1, that constitute a well established reference in the field. These algorithms incorporate the graph coloring routines of the software package *ColPack* [15, 16] and the sparsity detection and Hessian-vector product procedures of ADOL-C [30]. We shall denote them by the name of the coloring scheme employed: Star and Acyclic. Analytical properties of these algorithms, as well as numerical experiments with them, have been reported in [14, 30].

We have hand-picked fifteen functions from the CUTE collection [5] and one — augmlagn — from [22] for the experiments. The selection was based on the following criteria: Hessian's sparsity pattern, scalability and sparsity. We wanted to cover a variety of patterns; to be able to freely change the scale of the function, so as to appraise the performance of the algorithms as the dimension grows; and we wanted to work with sparse matrices. The appendix of [17] presents results for dimension values n in the set 5 000, 20 000, 50 000 and 100 000, but the tables in this section always refer to the $n = 50\,000$ case, unless otherwise explicitly noted.

The list of functions is presented in Table 5.1. The 'Pattern' column indicates the type of sparsity pattern: bandwidth[1] (B x), arrow, box, or irregular pattern. The last two display the number of columns of the *seed matrix* produced by Star and Acyclic, for dimension equal to 50 000. In order to report the performance of these algorithms, we briefly recall their *modus operandi*. Their first step, executed only once, computes a seed matrix S via coloring methods, such that the Hessian f'' may be recovered from the product $f''S$, which involves as many Hessian-vector products as the number of columns of S. The latter coincides with the number of colors used in the coloring of a graph model of the Hessian. The recovery of the Hessian boils down to the solution of a linear system. Thus the first computation of the Hessian takes necessarily longer, because it comprises two steps, where the first one involves

[1]The bandwidth of matrix $M = (m_{ij})$ is the maximum value of $|i - j|$ such that $m_{ij} \neq 0$.

Name	Pattern	# colors Star	Acyclic
cosine	B 1	3	2
chainwoo	B 2	3	3
bc4	B 1	3	2
cragglevy	B 1	3	2
pspdoc	B 2	5	3
scon1dls	B 2	5	3
morebv	B 2	5	3
augmlagn	5×5 diagonal blocks	5	5
lminsurf	B 5	11	6
brybnd	B 5	13	7
arwhead	arrow	2	2
nondquar	arrow + B 1	4	3
sinquad	frame + diagonal	3	3
bdqrtic	arrow + B 3	8	5
noncvxu2	irregular	12	7
ncvxbqp1	irregular	12	7

Table 5.1: Test functions

the coloring, and the second one deals with the calculation of the actual numerical entries. In subsequent Hessian computations, only the second step is executed, resulting in a shorter run. It should be noted that the number colors is practically insensitive to changes in the dimension of the function in the examples considered, with the exception of the functions with irregular patterns, noncvxu2 and ncvxbqp1.

Table 5.2 reports the times taken by `edge_pushing` and by the first and second Hessian computations by Star and Acyclic. It should be pointed out that Acyclic failed to recover the Hessian of ncvxbqp1, the last function in the table. In the examples where `edge_pushing` is faster than the second run of Star (resp., Acyclic), we can immediately conclude that `edge_pushing` is more efficient for that function, at that prescribed dimension. This was the case in 14 (resp., 16) examples. However, when the second run is faster than `edge_pushing`, the corresponding coloring method may eventually win, if the Hessians are computed a sufficient number of times, so as to compensate the initial time investment. This of course depends on the context in which the Hessian is used, say in a nonlinear optimization code. Thus the number of evaluations of Hessians is linked to the number of iterations of the code. The minimum time per example is highlighted in Table 5.2.

Focusing on the two-stage Hessian methods, we see that Star always has fastest second runtimes. Only for function sinquad is Star's first run faster than Acyclic's. Nevertheless, this higher investment in the first run is soon paid off, except for functions arwhead, nondquar and bdqrtic, where it would require over 1600, 50 and 25, respectively, computations of the Hessian to compensate the slower first run. We can also see from Tables 5.1 and 5.2 that

	Star		Acyclic		
Name	1st	2nd	1st	2nd	e_p
cosine	9.93	0.16	9.68	2.52	0.15
chainwoo	35.07	0.33	33.24	5.08	0.30
bc4	10.02	0.25	10.00	2.56	0.25
cragglevy	28.17	0.79	28.15	2.60	0.48
pspdoc	10.31	0.35	10.27	4.39	0.23
scon1dls	11.00	0.59	10.97	4.96	0.40
morebv	10.36	0.46	10.33	4.49	0.35
augmlagn	15.99	0.68	8.36	16.74	0.27
lminsurf	9.30	1.01	9.24	3.89	0.35
brybnd	11.87	2.44	11.73	12.63	1.68
arwhead	176.50	0.16	45.86	0.24	0.20
nondquar	166.59	0.18	28.64	2.57	0.12
sinquad	606.72	0.26	888.57	1.51	0.32
bdqrtic	262.64	1.34	96.87	7.80	0.80
noncvxu2	29.69	1.10	29.27	7.76	0.28
ncvxbqp1	13.51	2.42	–	–	0.37
Averages	87.98	0.78	82.08	5.32	0.41
Variances	25 083.44	0.54	50 313.10	19.32	0.14

Table 5.2: Runtimes in seconds for Star, Acyclic and `edge_pushing`.

Star's performance on the second run suffers the higher the number of colors needed to color the Hessian's graph model, which is to be expected. Thus the second runs of lminsurf, brybnd, bdqrtic, noncvxu2 and ncvxbqp1 were the slowest of Star's. Notice that, although the Hessian of bdqrtic doesn't require as many colors as the other four just mentioned, the function evaluation itself takes longer.

On a contrasting note, edge_pushing execution is not tied to sparsity patterns and thus this algorithm proved to be more robust, depending more on the density and number of nonlinear functions involved in the calculation. In fact, this is confirmed by looking at the variance of the runtimes for the three algorithms, see the last row of Table 5.2. Notice that edge_pushing has the smallest variance. Furthermore, although Star was slightly faster than edge_pushing in the second run for the functions arwhead and sinquad, the time spent in the first run was such that it would require over 4 000 and 10 000, respectively, evaluations of the Hessian to compensate for the slower first run.

The bar chart in Figure 5.4, built from the data in Table 5.2, permits a graphical comparison of the performances of Star and edge_pushing. Times for function brybnd deviate sharply from the remaining ones, it was a challenge for both methods. On the other hand, function ncvxbq1 presented difficulties to Star, but not to edge_pushing.

The bar chart containing the runtimes of the three algorithms is made pointless by the range of runtimes of Acyclic, much bigger than the other two. To circumvent this problem, we applied the base 10 log to the runtimes multiplied by 10 (just to make all logs positive). The resulting chart is depicted in Figure 5.5.

Although the results presented in Table 5.2 correspond to the dimension 50 000 case, they represented the general behavior of the algorithms in this set of functions. This is evidenced by the plots in Figures 5.6 and 5.7, that show the runtimes of edge_pushing and Star on four functions for dimensions varying from 5 000 to 100 000.

The functions cosine, sinquad, brybnd and noncvxu2 were selected for these plots because they exemplify the different phenomena we observed in the 50 000 case. For instance, the performances of both edge_pushing and Star are similar in the functions cosine and sinequad, and this has happened consistently in all dimensions. Thus the dashed and solid lines in Figure 5.6 intertwine, and there is no striking dominance of one algorithm over the other. Also, these functions presented no real challenges, and the runtimes in all dimensions are low.

The function brybnd was chosen because it presented a challenge to all methods, and ncvxu2 is the representative of the functions with irregular sparsity patterns. The plots in Figure 5.7 show a consistent superiority of edge_pushing over Star for these two functions. All plots are close to linear, with the exception of the runtimes of Star for the function noncvxu2. We observed that the number of colors used to color the graph model of its Hessian varied quite a bit, from 6 to 21. This highest number occurred precisely for the dimension 70 000, the most dissonant point in the series.

The appendix of [17] contains the runtimes for the three methods, including first and second runs, for all functions, for dimensions 5 000, 20 000 and 100 000.

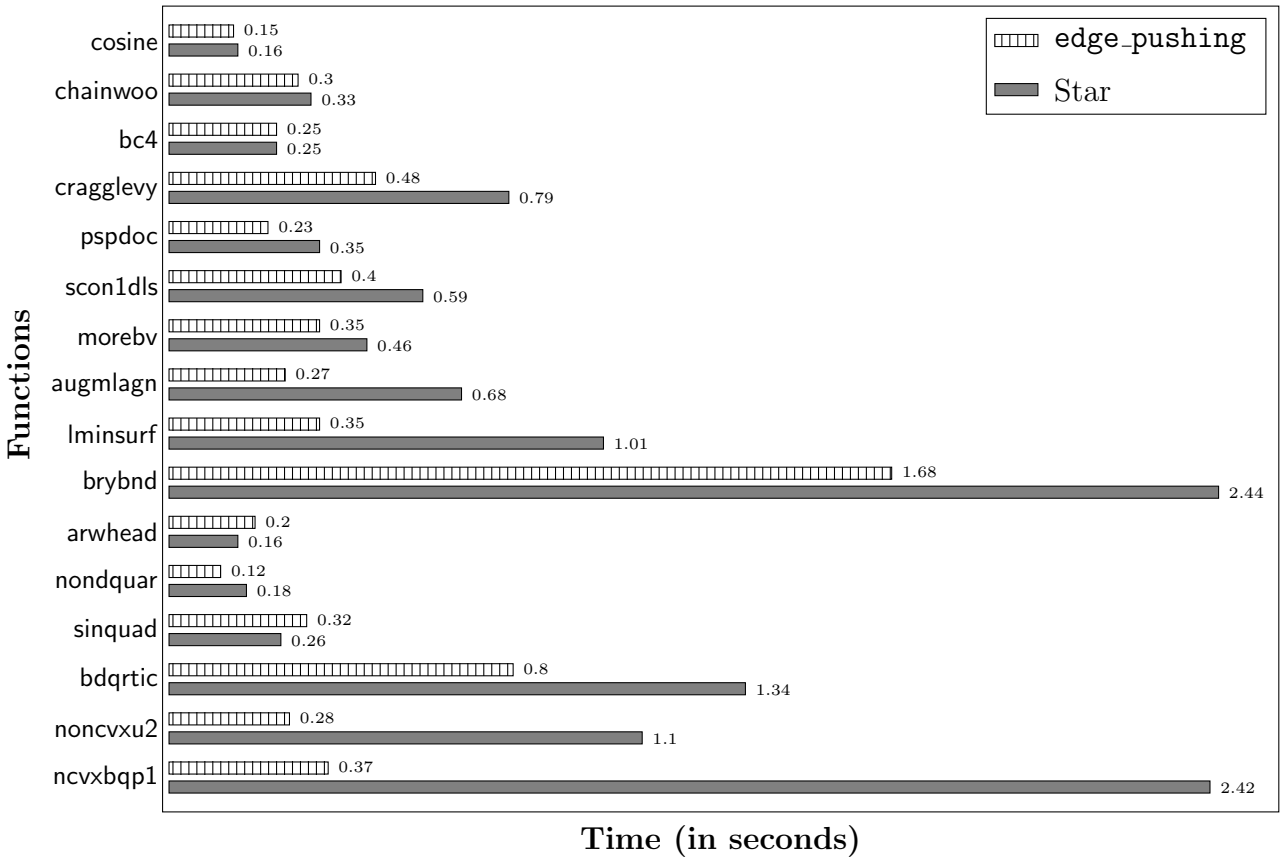


Figure 5.4: Graphical comparison: Star versus `edge_pushing`.

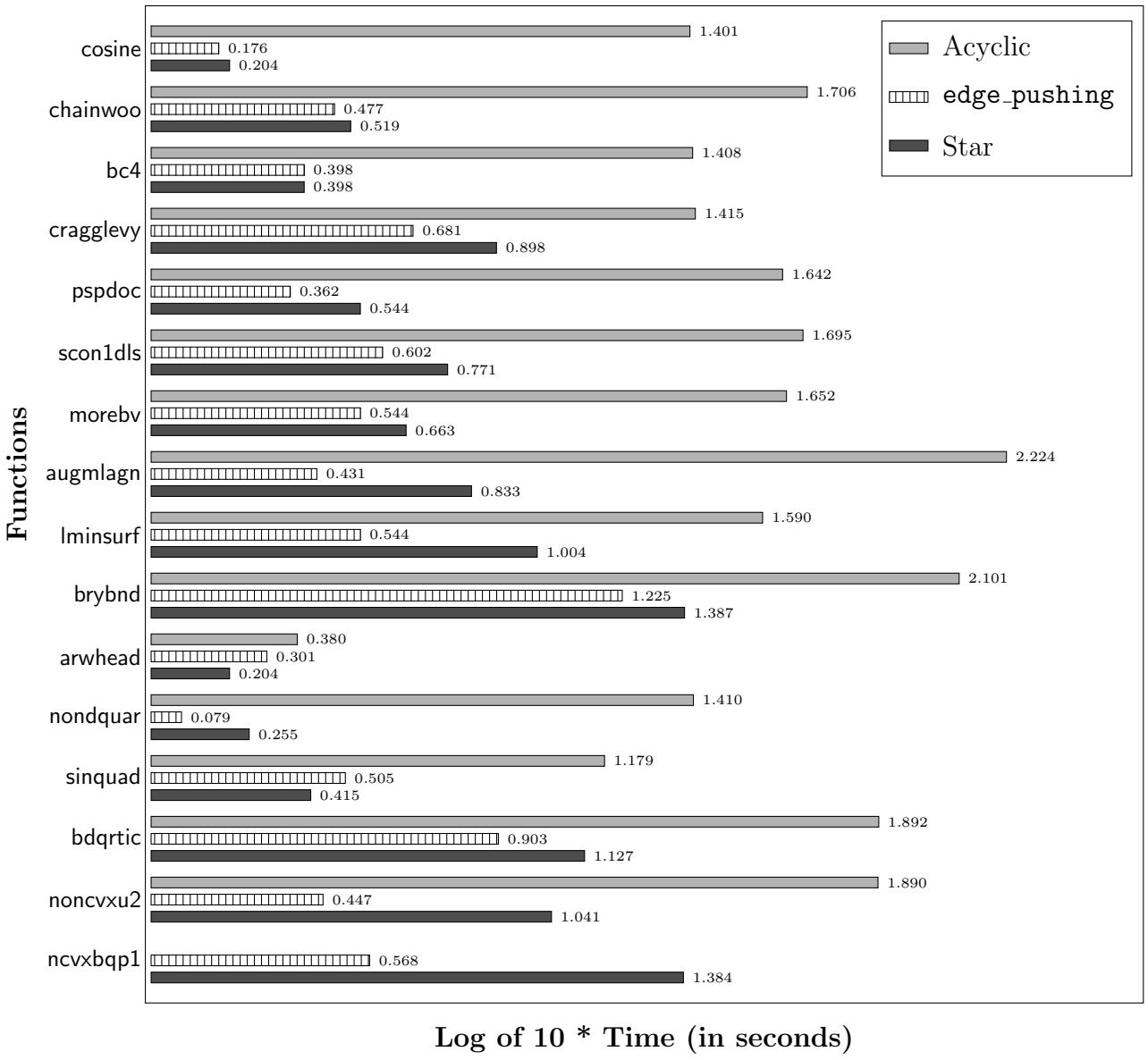


Figure 5.5: Graphical comparison of times in log scale: Star, Acyclic and `edge_pushing`.

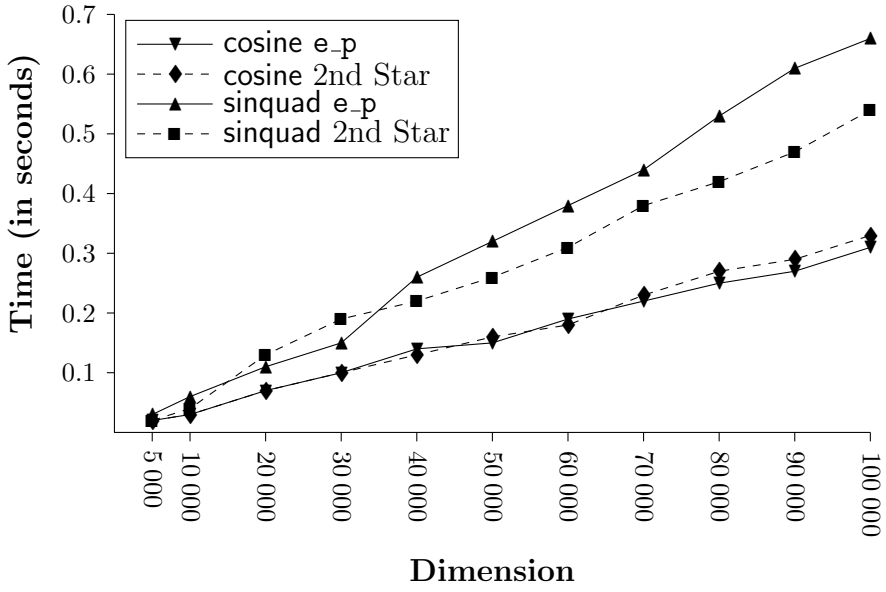


Figure 5.6: Evolution of runtimes of `edge_pushing` and Star (2nd run) with respect to dimension, for cosine and sinequad.

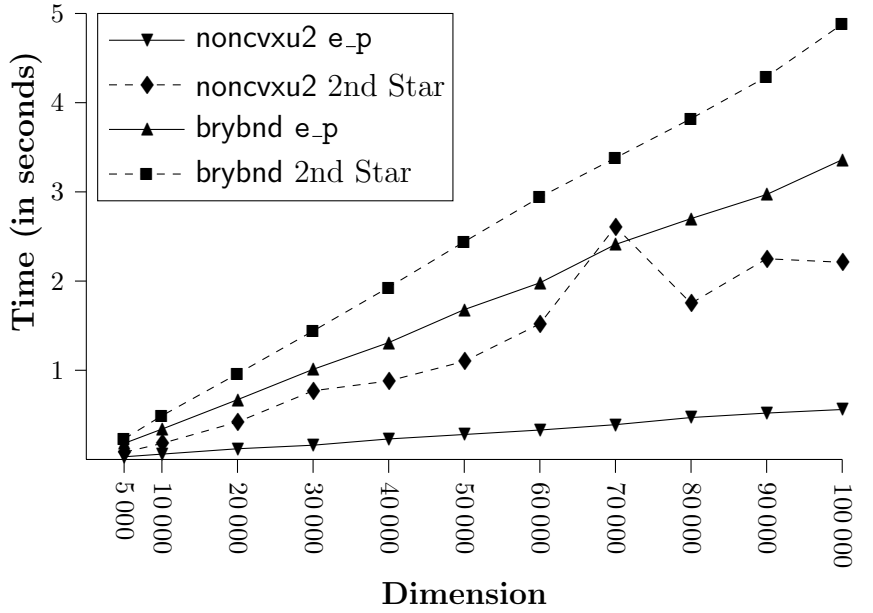


Figure 5.7: Evolution of runtimes of `edge_pushing` and Star (2nd run) with respect to dimension, for noncvxu2 and brybnd.

5.6 Discussion

The formula (4.17) for the Hessian obtained in Corollary 4.1 leads to new correctness proofs for existing Hessian computation algorithms and to the development of new ones. We also provided a graph model for the Hessian computation and both points of view inspired the construction of `edge_pushing`, a new algorithm for Hessian computation that conforms to Griewank and Walther's Rule 16 of Automatic Differentiation [20, p. 240]:

> *The calculation of gradients by nonincremental reverse makes the corresponding computational graph symmetric, a property that should be exploited and maintained in accumulating Hessians.*

The new method is a truly reverse algorithm that exploits the symmetry and sparsity of the Hessian. It is a one-phase algorithm, in the sense that there is no preparatory run where a sparsity pattern needs to be calculated that will be reused in all subsequent iterations. This can be an advantage if the function has a discontinious second derivative, thus requiring recalculation of the sparsity pattern when changing from one continouos region to another. For instance $h(u) = (\max\{-u, 0\})^2$. This type of function is used as a differentiable penalization of the negative axis. It is not uncommon to observe the 'thinning out' of Hessians over the course of nonlinear optimization, as the iterations converge to an optimum, which obviously lies in the feasible region. If the sparsity structure is fixed at the beginning, one cannot take advantaged of this slimming down of the Hessian.

`edge_pushing` was implemented as a driver of ADOL-C[19] and tested against two other algorithms, the Star and Acyclic methods of ColPack [16], also available as drivers of ADOL-C. Computational experiments were run on sixteen functions of the CUTE collection [5]. The results show the strong promise of the new algorithm. When compared to Star, there is a clear advantage of `edge_pushing` in fourteen out of the sixteen functions. In the remaining two the situation is unclear, since Star is a two-stage method and the first run can be very expensive. So even if its second run is faster than `edge_pushing`'s, one should take into account how many evaluations are needed in order to compensate the first run. The answers regarding the functions arwhead and sinquad were over 4 000 and 10 000, respectively, for dimension equal to 50 000. These numbers grow with the dimension. Finally, it should be noted that edge_pushing's performance was the more robust, and it wasn't affected by the lack of regularity in the Hessian's pattern.

We observed that Star was consistently better than Acyclic in all computational experiments. However, Gebremedhin et al. [14] point out that Acyclic was better than Star in randomly generated Hessians and the real-world power transmission problem reported therein, while the opposite was true for large scale banded Hessians. It is therefore mandatory to test `edge_pushing` not only on real-world functions, but also within the context of a real optimization problem. Only then can one get a true sense of the impact of using different algorithms for Hessian computation.

It should be pointed out that the structure of `edge_pushing` naturally lends itself to

parallelization, the topic of the final chapter. The opposite seems to be true for Star and Acyclic. The more efficient the first run is, the less colors, or columns of the seed matrix one has, and only the task of calculating the Hessian-vector products corresponding to $f''S$ can be seen to be easily parallelizable.

Appendix: Results for varying instance sizes

Dimension	5 000					20 000					100 000				
Name	Star		Acyclic		e_p	Star		Acyclic		e_p	Star		Acyclic		e_p
	1st	2nd	1st	2nd		1st	2nd	1st	2nd		1st	2nd	1st	2nd	
cosine	0.10	0.02	0.09	0.04	0.02	1.58	0.07	1.61	0.45	0.07	37	0.35	37	9.48	0.31
chainwoo	0.38	0.04	0.33	0.09	0.02	6.11	0.12	5.41	0.92	0.11	137	0.65	130	19.54	0.58
bc4	0.11	0.02	0.10	0.05	0.02	1.59	0.09	1.58	0.48	0.10	37	0.51	37	9.57	0.50
cragglevy	0.29	0.05	0.28	0.05	0.04	4.54	0.30	4.53	0.49	0.19	109	1.57	109	9.66	1.00
pspdoc	0.11	0.04	0.11	0.07	0.02	1.61	0.14	1.60	0.86	0.09	36	0.70	36	17.49	0.44
scon1dls	0.11	0.04	0.12	0.07	0.04	1.63	0.24	1.61	0.92	0.16	37	0.95	37	20.05	0.81
morebv	0.12	0.05	0.12	0.08	0.04	1.63	0.19	1.61	0.91	0.14	37	0.88	37	18.13	0.73
augmlagn	0.13	0.07	0.11	0.21	0.02	1.64	0.28	1.36	2.83	0.12	84	1.40	33	65.98	0.55
lminsurf	0.12	0.09	0.12	0.09	0.03	1.57	0.45	1.55	0.78	0.14	36	2.30	36	15.04	0.68
brybnd	0.17	0.23	0.16	0.22	0.18	1.96	0.96	1.88	2.20	0.67	39	4.88	39	42.05	3.36
arwhead	1.52	0.01	0.42	0.03	0.02	28.80	0.06	9.99	0.09	0.09	943	0.31	233	0.47	0.42
nondquar	1.29	0.01	0.21	0.04	0.01	23.19	0.08	3.49	0.48	0.05	1012	0.35	340	9.62	0.25
sinquad	2.79	0.02	5.09	0.05	0.03	60.97	0.11	99.54	0.33	0.13	3905	0.54	8961	5.14	0.66
bdqrtic	1.55	0.13	0.48	0.22	0.09	28.62	0.55	7.66	1.40	0.34	4323	2.68	833	71.4	1.65
noncvxu2	0.32	0.08	0.32	0.12	0.03	4.85	0.42	4.73	1.41	0.12	118	2.21	117	29.45	0.56
ncvxbqp1	0.15	0.20	–	–	0.02	2.22	0.91	–	–	0.13	51	5.39	–	–	0.77
Averages	0.58	0.07	0.54	0.10	0.04	10.78	0.31	9.88	0.97	0.17	684	1.60	734	22.9	0.83

Table 5.3: Runtimes for all methods and functions, at varying dimensions.

Chapter 6

Obtaining the Hessian's Sparsity Pattern using AD

Obtaining the sparsity pattern of the Hessian is a necessary step in a well-known method for calculating sparse Hessians using Automatic Differentiation or Finite Differences [13, 14]. With the sparsity pattern in hand, one may also use univariante Taylor series or second order scalar methods to individually calculate each nonzero element in the Hessian [1, 4].

We describe two methods for automatically calculating the sparsity pattern that use the framework of automatic differentiation, both of which consider a set of elemental functions composed of only unary and binary operations. The first method is by Andreas Walther [30] which has been implemented as the driver called `hess_pat` in ADOL-C [19]. Essentially `hess_pat` propagates nonlinearity information forward through the function evaluation. The second is a new method called `edge_pushing_sp` which is an algorithm adapted from the `edge_pushing` algorithm. `edge_pushing_sp` does the opposite in the sense that it propagates nonlinearity information in the reverse order. We compare the two algorithms using complexity analysis and numerical tests.

The sparsity structure that we are interested in is a structure that indicates the positions in the Hessian matrix that are not always zero. We refer to this sparsity structure as the sparsity super-structure.

Let $f : \mathbb{R}^n \to \mathbb{R}$ and $x \in \mathbb{R}^n$. The sparsity structure of the Hessian $f''(x)$ is a set of unordered pairs where:

$$\{j,k\} \in \text{ sparsity structure } \Leftrightarrow \frac{\partial^2 f}{\partial x_j \partial x_k}(x) \neq 0 \in \mathbb{R}.$$

The sparsity super-structure of the second derivative f'' is a set of unordered pairs where:

$$\{j,k\} \in \text{ sparsity super-structure } \Leftrightarrow \frac{\partial^2 f}{\partial x_j \partial x_k} \neq 0 \in C^2(\mathbb{R}).$$

The common idea behind `hess_pat` and `edge_pushing_sp` is that if $\partial^2 f/\partial x_j \partial x_k$ is not identically zero then there exists an intermediate variable $v_i = u_i(x)$ such that $\partial^2 u_i/\partial x_j \partial x_k$

is not identically zero. This is equivalent to saying that there exists r, s and i such that $\partial^2\phi_i/\partial v_r \partial v_s$ is not identically zero and $j \prec^* r$ and $k \prec^* s$, where $\prec^*$ is the transitive closure of the precedence relation. This motivates the definition of *nonlinear interactions*. We say that node j has a nonlinear interaction with node k if there exist i, r and s such that $\partial^2\phi_i/\partial v_r \partial v_s$ is not identically zero, $j \prec^* r$ and $k \prec^* s$.

One can build an over-estimate of the sparsity super-structure by including all pairs $\{j, k\}$ such that j has a nonlinear interaction with k. This is an overestimate due to degeneracy, for example consider the sequence of statements

$$v_1 = \sin(x_0),\ v_2 = \cos(x_0),\ v_3 = v_1 v_1,\ v_4 = v_2 v_2,\ v_5 = v_3 + v_4.$$

Our overestimate would include the pair $\{0, 0\}$ when in fact the sparsity super-structure of this evaluation procedure is empty. In practical examples, both methods calculate this over-estimated sparsity super-structure. For most function programs this overestimated sparsity super-structure is in fact the sparsity super-structure. From here on we drop the "overestimated" prefix for brevity.

6.1 Walther's Forward Mode: `hess_pat`

The first tool for calculating structural information of the Hessian automatically was implemented in AMPL [12]. By automatically detecting a partially separable structure, or, in other words, identifying that the function of interest is a sum of nonlinear terms, the Hessian matrices of the nonlinear terms are calculated separately and summed together to produce the desired Hessian matrix. Although this procedure exploits the sparsity pattern, is does not calculate it.

An algorithm that calculates the actual sparsity structure of the Hessian was proposed and analyzed in 2008 by Walther [30]. Roughly speaking, the algorithm walks through the function's list of intermediate variables and upon encountering a nonlinear elemental function ϕ_i it checks to see how ϕ_i contributes to the sparsity super-structure of the Hessian. To do due this, the linear dependencies between intermediate variables and independent variables are stored in ℓ sets χ_i called index domains, defined in [20] as follows

$$\chi_i := \{j \leq n : j - n \prec^* i\} \quad \text{for } 1 \leq i \leq \ell.$$

Each χ_i contains the indices of the independent variables necessary to calculate v_i, in other words, there exists $u : \mathbb{R}^{|\chi_i|} \to \mathbb{R}$ such that $u(x) = v_i$.

If we also create index domains for the independent variables and set $\chi_{j-n} := \{j\}$ for $1 \leq j \leq n$, one can compute the index domains using the forward recurrence:

$$\chi_i \leftarrow \bigcup_{j \prec i} \chi_j, \quad \text{for } i = 1, \ldots, \ell.$$

To keep track of the accumulating sparsity super-structure, Walther [30] defines the *Nonlinear interaction domains* $\mathcal{N}_j, j = 1, \ldots, n$,

$$\mathcal{N}_j := \{k \leq n \ : \ j \text{ has a nonlinear interaction with } k\}.$$

The pseudo code of `hess_pat` is in Algorithm 6.1. Upon execution, the algorithm preforms a forward sweep of the intermediate variables from v_1 to v_ℓ. At the ith iteration, it calculates χ_i by merging the index domains of v_i's predecessors. If ϕ_i is a nonlinear function, the *nonlinear interaction domain* of the independent variables indexed in χ_i must be updated. How they are updated depends on the type of nonlinear function ϕ_i is: If it is unary $\phi_i(v_j)_{j \prec i} = \phi_i(v_j)$ then every variable indexed in χ_i has a nonlinear interaction with every other variable therein. To carry-out this update we merge $\mathcal{N}_p$ for every $p \in \chi_i$ with χ_i, see line (8). If ϕ_i is binary $\phi_i(v_j)_{j \prec i} = \phi_i(v_j, v_k)$ and nonlinear there are two possibilities: i) either ϕ_i is bilinear therefore each variable indexed in χ_j has a nonlinear interaction with each indexed in χ_k, lines (11) and (15), or ii) it is nonlinear in both v_j and v_k and every variable indexed in χ_i has a nonlinear interaction with every other variable therein, lines (13) and (17).

Algorithm 6.1: Walther's Forward Mode: `hess_pat`

Input: A taped evaluation procedure $\mathcal{T}$ of $f(x)$

```
1  for i = 1, ... n do
2      χ_{i-n} ← {i}, N_i ← ∅
3  end
4  for i = 1, ... ℓ do
5      χ_i ← ⋃_{j≺i} χ_j
6      if φ_i is nonlinear then
7          if v_i = φ_i(v_j) then
8              ∀p ∈ χ_i : N_p ← N_p ∪ χ_i
9          if v_i = φ_i(v_j, v_k) then
10             if φ_i is linear in v_j then
11                 ∀p ∈ χ_j : N_p ← N_p ∪ χ_k
12             else
13                 ∀p ∈ χ_j : N_p ← N_p ∪ χ_i
14             if φ_i is linear in v_k then
15                 ∀p ∈ χ_k : N_p ← N_p ∪ χ_j
16             else
17                 ∀p ∈ χ_k : N_p ← N_p ∪ χ_i
18 end
```

Output: A matrix sparsity pattern, where ith row contains the indices in $\mathcal{N}_i$.

The algorithm has been implemented and incorporated to ADOL-C [19] as a driver called `hess_pat`. We will use this driver later on for comparative numerical tests.

6.2 A New Reverse Mode: `edge_pushing_sp`

From the `edge_pushing` Algorithm 4.9 we can extract an algorithm for calculating the sparsity super-structure. This method for calculating the sparsity structure sweeps through

the evaluation procedure in the reverse order. Upon encountering a nonlinear function ϕ_i such that $\partial^2\phi_i/\partial v_j \partial v_k \neq 0$, an edge is allocated incident to nodes j and k to represent the nonlinear interaction between v_j and v_k. This nonlinear dependency is "pushed down" to predecessors by pushing and splitting the edge. Contrasting to `hess_pat`, the contribution of a nonlinear function ϕ_i to the sparsity pattern is not immediately calculated. Instead, the nonlinear function initiates a trickle of edges down the computational graph. These edges will eventually connect independent variables, and an edge connecting x_j to x_k means that $\{j, k\}$ is in the overestimated sparsity super-structure. We call the algorithm `edge_pushing_sp`. To transform `edge_pushing` into this sparsity calculating algorithm, we dispensed of all things related to correctly calculating weights of edges and simply maintain edges that would have weights that are not always zero.

The resulting `edge_pushing_sp` pseudo code is in Algorithm 6.2. The input for `edge_pushing_sp` is a recorded evaluation procedure $\mathcal{T}$, but differently from the usual recorded evaluation procedure for reverse routines such as the `edge_pushing` Algorithm 4.9, the floating point values of v_i's need not be recorded.

6.3 Complexity Bounds

6.3.1 Bounds for Partially Separable Functions

Our objective here is to argue that, for practical cases, `edge_pushing_sp` is more suited for obtaining the sparsity pattern of partially separable functions (5.2) then `hess_pat`. Simply put, the time spent in merging the index domains on line (5) of the `hess_pat` Algorithm 6.1 is costly. Specifically, under reasonable assumptions, the temporal complexity of calculating the index domains is bounded below by $\Omega(n^2/p + m)$. While `edge_pushing_sp` is bounded above by $O(mp^2q^3)$. We will emphasize these bounds with numerical tests on partially separable functions in which we can grow n. From this point on, we prove these claims.

When using operator overloading to generate an evaluation procedure, the format of the computational graph in Figure 6.3.1 is very common for partially separable functions. We define such evaluation procedure as a *partially separable O.O.P* (Operator Overloading Procedure). Each node f_i in Figure 6.3.1, is the root of a computational graph that calculates the nonlinear function $f_i(x_{I_i})$. The v_j nodes are the partial sums:

$$v_j = \sum_{i=1}^{j} f_i(x_{I_i}), \;\; j = 1, \ldots, m-1.$$

Such a computational graph is generated when the function program supplied is like the one in Algorithm 6.3. This is truly a natural way of computing a partially separable function, so much so, almost all of our test functions from the CUTE collection possess such a format. The few that were not in this format, could be made so with small adjustments. However, no adjustments were made on the functions for the computational tests.

Algorithm 6.2: `edge_push_sp`: Sparsity Pattern calculation.

Input: A taped evaluation procedure $\mathcal{T}$ of $f(x)$
Initialization: a graph $G = (V, E)$ with $|V| = n + \ell$ and $E = \emptyset$
for $i = \ell : 1$ **do**
  `Pushing:`
  foreach $\{i, p\} \in E$ **do**
    if $v_i = \phi_i(v_j, v_k)$ **then**
      $E \leftarrow E \cup \{p, j\}$
      $E \leftarrow E \cup \{p, k\}$
    else if $v_i = \phi_i(v_j)$ **then**
      $E \leftarrow E \cup \{p, j\}$
  end
  `Creating:`
  if ϕ_i *is nonlinear* **then**
    if $v_i = \phi_i(v_j)$ **then**
      if $\dfrac{\partial^2 \phi_i}{\partial v_j^2} \neq 0$ **then**
        $E \leftarrow E \cup \{j, j\}$
    else if $v_i = \phi_i(v_j, v_k)$ **then**
      if $\dfrac{\partial^2 \phi_i}{\partial v_j^2} \neq 0$ **then**
        $E \leftarrow E \cup \{j, j\}$
      if $\dfrac{\partial^2 \phi_i}{\partial v_j \partial v_k} \neq 0$ **then**
        $E \leftarrow E \cup \{k, j\}$
      if $\dfrac{\partial^2 \phi_i}{\partial v_k^2} \neq 0$ **then**
        $E \leftarrow E \cup \{k, k\}$
end
Output: The subgraph of G formed by the node set $\{1 - n, \ldots, 0\}$.

Algorithm 6.3: A typical function program of a partially separable function $f(x) = y$.

Input: $x \in \mathbb{R}^n$
Intialization: $y = 0$
for $i = 1 : m$ **do**
  $y = y + f_i(x_{I_i})$
end
Output: y

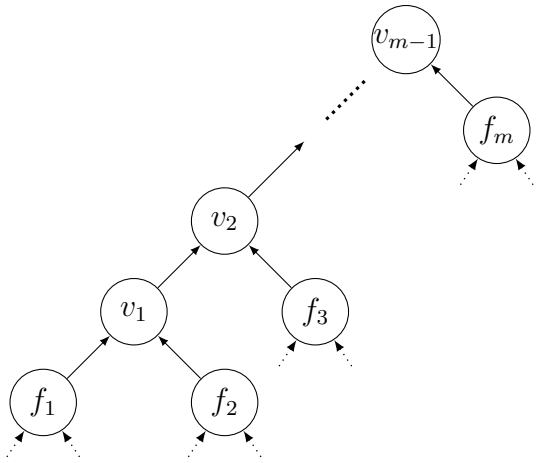


Figure 6.1: Computational graph of a partially separable O.O.P

Proposition 6.1 *The temporal complexity of calculating the index domains χ_i, $i = 1, \ldots, \ell$, of partially separable O.O.P is $\Omega\left(n^2/p + m\right)$.*

Demonstration: To prove Proposition 6.1, first note that independently of the merging method used, the operation count of merging two ordered sets is at least the sum of their respective cardinalities.[1]

Let χ_j be the index domain of the partial sum variable v_j, $j = 1, \ldots, m-1$. Let χ_{fi} be the index domain of the root variables f_i, $i = 1, \ldots, m$. We use p to denote the maximum number of independent variables indexed in any given χ_{fi}, thus $|\chi_{fi}| \leq p$, for $i = 1, \ldots, m$.

We will obtain a lower bound for calculating the index domains by bounding the complexity of computing each χ_j, $j = 1, \ldots, m$. The set χ_1 is built by merging χ_{f_1} and χ_{f_2}, while χ_j are built by merging the sets χ_{j-1} and $\chi_{f_{j+1}}$, for $j = 2, \ldots, m$. Necessarily $|\chi_{m-1}| = n$, hence there are at least n operation counts in merging χ_{m-2} and χ_{f_m}. Given there are at most p independent variables in the graph rooted by f_m, we have that $|\chi_{m-2}| \geq n - p$ which in turn incurs an operation count of at least $n - p$ in calculating the set χ_{m-2}. By induction we have:

$$\text{Operation Count}(\chi_{m-j}) \geq \max\{0, n - (j-1)p\}.$$

Thus, for the total number of operation counts in calculating all χ_j, we have the following

[1]This of course excludes cases where we have specific knowledge of the sets.

bound:

$$\begin{aligned}\sum_{j=1}^{m} \text{TIME}(\chi_{m-j}) &\geq n + (n-p) + \cdots + (n - \left\lfloor \frac{n}{p} \right\rfloor p) + \underbrace{1 + \cdots + 1}_{m - \lfloor n/p \rfloor} \\ &= \frac{n}{2}\left\lfloor \frac{n}{p} \right\rfloor + m - \left\lfloor \frac{n}{p} \right\rfloor \\ &= \left(\frac{n}{2} - 1\right)\left\lfloor \frac{n}{p} \right\rfloor + m.\end{aligned}$$

Hence the operation count for calculating the index domains is $\Omega(n^2/p + m)$. ∎

The upper bound $O(mp^2q^3)$ is immediate for `edge_pushing_sp` given that it has the same complexity of `edge_pushing` Algorithm 5.2, and we proved this bound in Proposition 5.4 for all partially separable functions such that the last $m-1$ intermediate variables are sums.

6.4 Computational Experiments

All tests were run on the 32-bit operating system Ubuntu 9.10, processor Intel 2.8 GHz, and 4 GB of RAM. All algorithms were coded in C and C++. Both `edge_pushing_sp` and `hess_pat` algorithms have been implemented as drivers of ADOL-C, and use the same taped evaluation procedure produced by ADOL-C [19]. The output of the two algorithms is the same overestimated sparsity super-structure.

As test cases, we have chosen the Lagrange function of the CUTE problems `broydnbd`, `chainwoo`, `lminsurf`, `morebv` and `sinquad` with varying dimension n. The first four functions are precisely the test functions used in [30]. We chose the last function `sinquad` for its Hessian structure has a property that the others do not: a dense row. The runtimes are displayed in Table 6.4. To emphasize the asymptotic behavior, we vary dimension from 1000 to 2×10^5. All test cases are partially separable with m being linear in n, whereas p and q were independent of n. The nonlinear dependence on n of the execution time of `hess_pat`, Proposition 6.1, becomes apparent. All five examples have similar results, hence we have plotted only one of them, `morebv`, in Figure 6.2.

Accumulating linear dependencies between intermediate variables and independent variables, as is done in `hess_pat`, becomes costly as the dimension of the problem grows. This might well be a necessary aspect of algorithms that perform a forward sweep. While accumulating nonlinear dependencies in a reverse sweep, we need not carry these linear dependencies. Instead, only known contributions to the sparsity pattern are dealt with.

As an immediate application of `edge_push_sp`, one can make the Hessian methods that use graph colouring [13, 14] more competitive.

	morebv		chainwoo		lminsurf		brydnbd		sinquad	
n	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp	hess_p	e_p_sp
1000	0.03	0.01	0.02	0.02	0.01	0.01	0.01	0.03	0.01	0.01
2000	0.05	0.02	0.05	0.02	0.02	0.01	0.03	0.06	0.05	0.02
4000	0.06	0.05	0.21	0.02	0.06	0.02	0.08	0.12	0.20	0.02
6000	0.13	0.05	0.41	0.02	0.13	0.02	0.17	0.18	0.44	0.03
8000	0.24	0.05	0.84	0.03	0.24	0.03	0.28	0.25	0.77	0.04
10000	0.38	0.07	1.31	0.04	0.38	0.04	0.42	0.29	1.21	0.05
12000	0.54	0.09	1.97	0.04	0.52	0.06	0.59	0.34	1.74	0.06
16000	0.94	0.11	3.37	0.07	0.93	0.08	1.01	0.49	3.10	0.08
20000	1.61	0.14	5.33	0.09	1.42	0.10	1.56	0.58	4.88	0.10
30000	3.29	0.21	12.06	0.12	3.22	0.15	3.39	0.88	11.62	0.15
40000	6.02	0.27	21.20	0.16	5.74	0.20	6.03	1.15	32.64	0.20
60000	13.74	0.41	47.83	0.24	12.70	0.30	13.27	1.86	106.02	0.29
80000	23.68	0.57	83.20	0.32	22.59	0.39	23.25	2.33	214.99	0.38
100000	36.73	0.72	131.03	0.41	35.94	0.49	38.88	3.06	359.33	0.49
200000	166.81	1.34	604.86	0.81	165.47	1.03	167.09	5.96	1532.26	1.02

Table 6.1: Runtime results for `edge_pushing_sp` (abbreviated to `e_p_sp`) and `hess_pat` (abbreviated to `hess_p`) over varying dimensions.

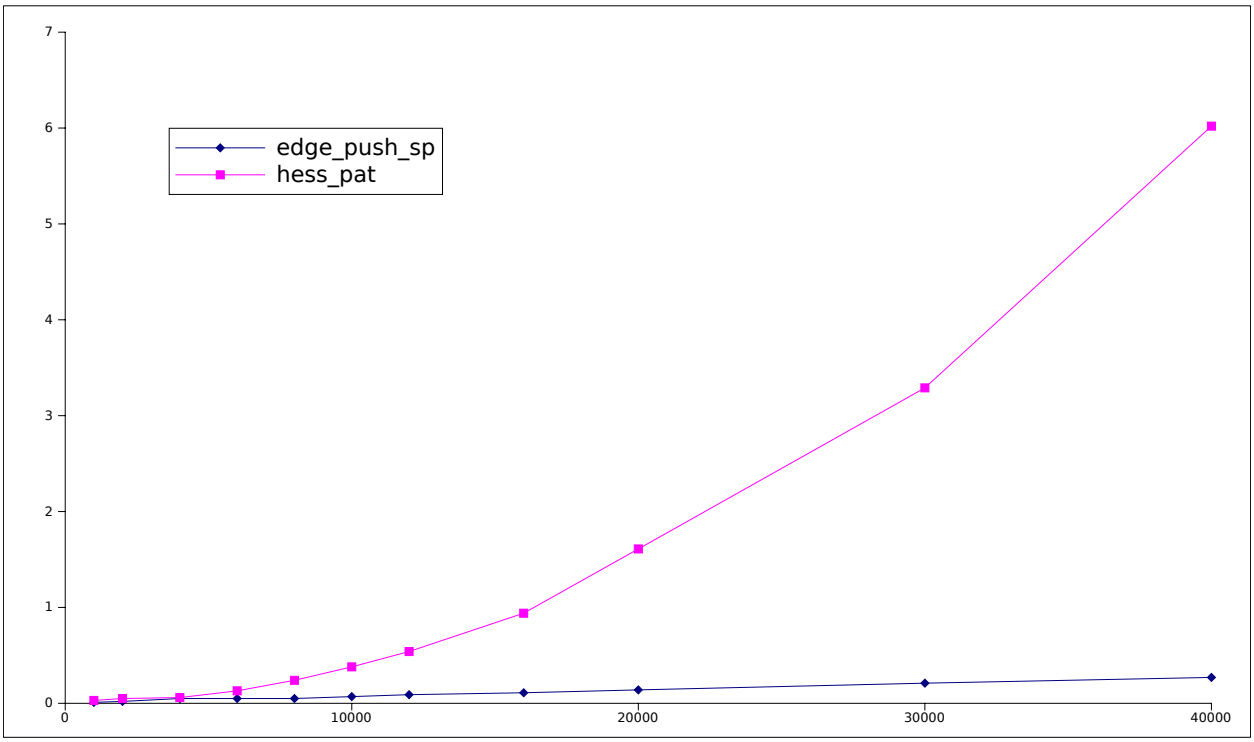


Figure 6.2: The time in seconds over varying dimension on problem `morebv`

Chapter 7

Parallel Sparse Hessian Computation

7.1 Introduction

Physical limitations that include temperature and a minimum transistor size combined with the growing cost of producing faster single processors are pushing parallel computing forward. Supercomputers such as can be found at `http://top500.org/` have been using concurrent processing since the '70s to solve large scientific and industrial problems, but now multi-core processors and parallel processing has become commonplace, with most all laptops on the market with two or more processing units.[1] It is desirable to take full advantage of this processing power, and efforts are being made throughout the computing community to adapt existing algorithms to make good use of this surge of parallel processing computers.

Functions that possess sparse Hessian matrices have a partially separable structure (5.2) such that $p \ll n$, [27]. One can use the partially separability of a function to break up the work load in calculating the Hessian. Due to the linearity of the differential operator, the Hessian of the partially separable function $f(x)$ in (5.2) is a summation of other Hessians:

$$H_f(x) = \sum_{i=1}^{m} H_{f_i}(x_{I_i}). \tag{7.1}$$

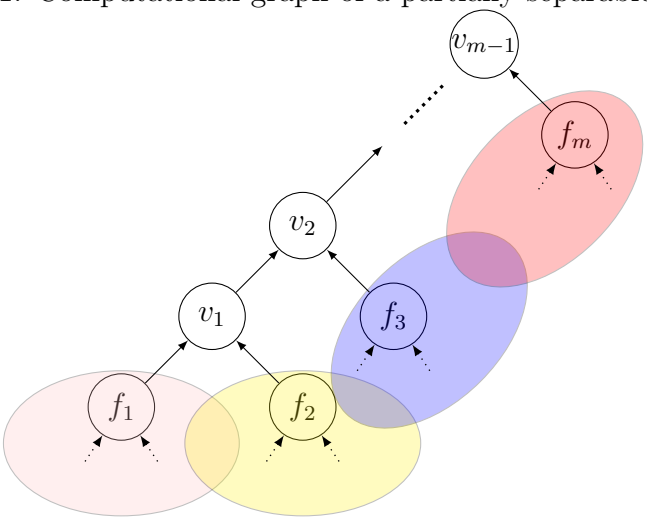
A simple method for dividing up the work is to assign to each processor the job of calculating a number of H_{f_i} Hessians. The computational graph in Figure 7.1 is compatible with the evaluation of a partially separable function. Each node f_i is the root of a computational graph that calculates the nonlinear function $f_i(x_{I_i})$. Each coloured ellipse represents a subgraph. The ellipses intersects to indicate that each graph rooted by a f_i may share nodes and independent variables with other subgraphs. The v_j nodes are the partial sums:

$$v_j = \sum_{i=1}^{j} f_i(x_{I_i}). \tag{7.2}$$

Though the total processing can be increased due to intersections in these subgraphs, one

[1] The laptops used to write this dissertation both had dual-core processors.

Figure 7.1: Computational graph of a partially separable function.



can still obtain speed-up through concurrent processing.

A general strategy for calculating the Hessian in parallel would be to cut up the taped evaluation procedure of a partially separable function, creating m severed pieces which correspond to the m f_i functions. Each processing unit then calculates the Hessian H_{f_i} associated to a number of f_i tapes, to finally send the separate Hessians to a single master processor that will merge the Hessians into one.

If one were to use the `edge_pushing` algorithm 5.2, or any other completely reverse Hessian routine, one would not need to cut up the tape previously, but simply distribute the original tape to all processing units, and allow each processor to calculate a number of H_{f_i} Hessians.

The strategies described here work for both distributed and shared memory models. In shared memory systems, one can concurrently execute different *threads* of tasks, in other words, instructions that are independent of one another. Though the separate threads are executed concurrently, the threads share cache memory. On the other hand, in a distributed system separate processors with separate memory can work concurrently to achieve a common goal. Though the memory is not shared between processors, which allows for simultaneous memory access, now interprocess communication costs become the issue. Throughout this chapter we will use threads to designate the separate tasks being executed, but keep in mind the strategies presented here can be implemented in a shared or distributed system. Now we work out the details of these two strategies, first the general, then the specific `edge_pushing` approach.

7.2 General approach

First we describe a sequential algorithm for cutting up a taped evaluation procedure of a partially separable function. After the pieces are severed, various threads may simultaneously calculate the Hessians associated with the severed pieces. In addition to the function being partially separable, assume that it is coded in such a way that the partial sum variables (7.2) are the last $m-1$ nodes of the computational graph.

7.2.1 Sequential Tape Cutting

The `tape_cut` Algorithm 7.1 receives as an input a taped evaluation procedure $\mathcal{T}$ and an integer `nthreads`: the number of threads that will concurrently calculate the Hessian. Its output is a sequence of tapes $\mathcal{T}_1 \dots \mathcal{T}_{\texttt{nthreads}}$. Algorithm 7.1 performs a reverse sweep of the original tape. The central idea behind the cutting algorithm is to transverse the partial sum variables, and by and large ignore them, to subsequently reach and identify the head of each f_i subgraph in Figure 7.1, then *flag* this head node as belonging to a certain processor. Specifically, processors are numbered 1 to `nthreads`, and during the `Setting Flags` step, the f_i head nodes are flagged to processors according to the congruence classes of mod (`nthreads`). Then in the `Passing Flags` step, flags are passed down to all predecessors of the head node, indicating that they belong to the same threads as their successors. To this end, lists $\texttt{flags}_i$ are created for $i = 1-n, \dots, \ell$. As node i is being swept, each $k \in \texttt{flags}_i$ indicates that node i has been flagged for the kth thread. Flags are passed down to predecessors $j \prec i$ by merging the set $\texttt{flags}_i$ into $\texttt{flags}_j$. After a node has passed on each flag $k \in \texttt{flags}_i$, its number and relevant information is recorded to the tape $\mathcal{T}_k$.

Aside from this central idea, a small precaution must be taken to avoid copying "dead end" nodes, that is, nodes that do not contribute to the end value of v_ℓ. To avoid assigning a dead end subgraph to a thread, all predecessors of the end node ℓ are marked as "belonging" by setting $\texttt{belongs}_j = 1$, for $j \prec \ell$. Then, as node i is visited in the reverse sweep, if $\texttt{belongs}_i = 1$ then the predecessors of i are also marked as belonging. By simply ignoring nodes that do not belong, dead ends are forefended.

To analyze the Sequential `tape_cut` Algorithm (7.1)'s complexity, first we fix the $\texttt{flags}_i$ sets, for $i = n-1, \dots, \ell$, as linked lists and restrict the set of elemental functions to unary and binary functions. As node i is swept, during the `Setting Flags` step, the inserting of elements into $\texttt{flags}_i$ need not be done in order, thus it can be carried out in constant time. Consequently so can the `Setting Flags` step be done in constant time.

In the `Passing Flags` step, for each $j \prec i$, the sets $\texttt{flags}_i$ and $\texttt{flags}_j$ are merged, thus the complexity of `Passing Flags` step depends on how many elements are in these sets. Given that there are `nthreads` threads, there can be at most `nthreads` integers in any set $\texttt{flags}_k$, for $k = 1-n, \dots, \ell$. Therefore the cost of merging any two `flags`. set is bounded by $O(\texttt{nthreads})$ and consequently the complexity of the `Passing Flags` step is bounded by $O(\texttt{nthreads})$. Hence each iteration of the Sequential `tape_cut` Algorithm 7.1 is bounded by $O(\texttt{nthreads})$, and the total complexity is $O(\texttt{nthreads}\ell)$.

Algorithm 7.1: Sequential `tape_cut`

Input: $\mathcal{T}$, `nthreads`
Initialization `cur_pro` =0, belongs$_i = 0$, flags$_i = \emptyset$, for $i = 1 - n \ldots \ell$,
tapes $\mathcal{T}_i = \emptyset$ for $i = 1 \ldots$ `nthreads`
for $i = 1, \ldots, \ell$ **do**
  if $\phi_i = v_j + v_k$ *and* $flags_i = \emptyset$ **then**
    belongs$_j$ =1
    belongs$_k$ =1
  else
    `Setting Flags`:
    if $flags_i = \emptyset$ *and* $belongs_i = 1$ **then**
      flags$_i \leftarrow$ `cur_pro`
      `cur_pro` = `cur_pro`+1
      if *cur_pro* $>$ *nthreads* **then**
        `cur_pro`=0
    `Passing Flags`:
    foreach $k \in flags_i$ **do**
      foreach $j \prec i$ **do**
        flags$_j$ = flags$_j \cup$ flags$_i$
      end
      foreach $k \in flags_i$ **do**
        $\mathcal{T}_k \leftarrow$ node i data
      end
    end
  end
end
Output: tapes $\mathcal{T}_1 \ldots \mathcal{T}_{\texttt{nthreads}}$

7.2.2 Parallel Tape Cutting

Though the `tape_cut` Algorithm 7.1 is a preparatory stage for parallel execution of Hessian algorithms, it can be done in parallel. By allowing all threads access to the original tape[2], each thread will simultaneously cut out a designated number of f_i functions from the original tape. Thus the input to the Parallel `tape_cut` Algorithm 7.2 is the original tape, the integer `nthreads` and an integer `mythread` that corresponds to which thread is executing the Parallel `tape_cut` Algorithm 7.2. Two binary vectors initially set to zero, `myflag` and `enemyflag`, with elements numbered $1 - n$ to ℓ are used to flag the nodes. As the tape is transversed in reverse order, the f_i head nodes encountered such that

$$\texttt{mythread} = i \mod \texttt{nthreads},$$

are flagged by marking $\texttt{myflag}_i = 1$, otherwise it is flagged as belonging to another thread by marking $\texttt{enemyflag}_i = 1$. In the `Passing Flags` step, flags of both types are passed down to predecessors, but only nodes flagged by the `myflag` vector are copied into the output tape $\hat{\mathcal{T}}$.

The complexity of the Parallel `tape_cut` Algorithm 7.2 is rather simple, for as node i is swept, there are no loops through vectors or lists, and by restricting the set of elemental functions to binary and unary functions, the overall complexity of Algorithm 7.2 is $O(\ell)$. The parallel tape cutting strategy consists of running the Parallel `tape_cut` Algorithm 7.2 simultaneously on `nthreads` threads. In a shared memory model, different threads cannot alter and access memory simultaneously. Thus, the allocating of `nthreads` sets of binary vectors `myflag` and `enemyflag` should be done sequentially, which can incur in a large execution time.

7.3 Parallel `edge_pushing`

With a completely reverse Hessian routine, one need not explicitly cut the tape, but simply deliver the original tape to all processors and restrict which H_{f_i} Hessians of (7.1) each processing unit is allowed to calculate. The general outline for a shared memory implementation is Algorithm 7.3, where the Hessian method adopted is `edge_pushing`. The commands followed by the hash # symbol indicate that the iterations of the following loop should be distributed amongst the threads. This style of commands for parallel execution is borrowed from the application programming interface of OpenMP [24]. Each call to `edge_pushing_p` accompanies a tape and an integer `thread`, and will only calculate the Hessians H_{f_i} such that

$$\texttt{mythread} = i \mod \texttt{nthreads}.$$

The output of `edge_pushing_p` is a graph structure that stores a Hessian matrix. After all calls to `edge_pushing_p` are finished, all graphs G_i, for $i = 1 \ldots \texttt{nthreads}$, are merged into one, which stores the desired Hessian matrix.

[2]In a distributed system this would be done by broadcasting the tape to all CPUs.

Algorithm 7.2: Parallel `tape_cut`

Input: $\mathcal{T}$, `nthreads`, `mythread`
Initialization $\text{belongs}_i = 0$, $\texttt{enemyflag}_i = 0$, $\texttt{myflag}_i = 0$, for $i = 1 - n \ldots \ell$,
`cur_pro` =0, tape $\hat{\mathcal{T}} = \emptyset$
for $i = 1, \ldots, \ell$ **do**
  if $\phi_i = v_j + v_k$ $\mathit{myflag}_i = 0$ *and* $\mathit{enemyflag}_i = 0$ **then**
    belongs_j =1
    belongs_k =1
  else
    `Setting Flags`:
    if $\mathit{myflag}_i = 0$ *and* $\mathit{enemyflag}_i = 0$ *and* $\mathit{belongs}_i = 1$ **then**
      if $\mathit{cur\_pro} = \mathit{mythread}$ **then**
        $\texttt{myflag}_i = 1$;
      else
        $\texttt{enemyflag}_i = 1$
      `cur_pro` = `cur_pro`+1
      if $\mathit{cur\_pro} > \mathit{nthreads}$ **then**
        `cur_pro`=0
    `Passing Flags`:
    if $\mathit{myflag}_i = 1$ **then**
      foreach $j \prec i$ **do**
        $\texttt{myflag}_j = 1$
      end
      $\hat{\mathcal{T}} \leftarrow$ node i data
    if $\mathit{enemyflag}_i = 1$ **then**
      foreach $j \prec i$ **do**
        $\texttt{enemyflag}_j = 1$
      end
  end
end
Output: tape $\hat{\mathcal{T}}$

Algorithm 7.3: Parallel `edge_pushing` strategy

Input: $\mathcal{T}$, `nthreads`
Initialization Graph G_i, $i = 1 \ldots$ nthreads, Graph G
#Begin Parallel For Loop region
for $thread = 1 \ldots nthreads$ **do**
    $G_i =$ `edge_pushing_p`($\mathcal{T}$, `thread`);
end
#End Parallel region
for $i = 1 \ldots nthreads$ **do**
    $G \leftarrow G \cup G_i$
end
Output: G

The code for `edge_pushing_p` is in Algorithm 7.4, and it is literally a merger between the Parallel `tape_cut` Algorithm 7.2 and `edge_pushing`. All the details of the Hessian calculation in Algorithm 7.4 are left out, and only the major steps: `Pushing`$_i$, `Creating`$_i$ and `Adjoint`$_i$ are indicated. The focus is on how one selects which nodes will enter into the calculations, and which will not. The same flagging procedure from Algorithm 7.2 is adopted, with the binary vector `myflag` for flagging nodes associated to H_{f_i} Hessians that will be calculated, and `enemyflag` for those that will not enter in the calculations.

The nodes encountered during the reverse sweep with nonlinear functions that have not been flagged are precisely the f_i head nodes. Therefore they are flagged in the `Setting Flags` step. Control over what nodes enter in the calculation is exerted by restricting the `Creating` step, which is only executed on nodes that have been flagged by `myflag`. Thus, the larger `nthreads` is, the fewer sequences of trickling down edges are initiated by the `Creating` step.

7.4 Concurrent Execution and Results

We have implemented the Parallel `edge_pushing` strategy for execution in a shared memory computer. The computer used for our experiments is a Mac Pro Xeon 64-bit workstation, with two 2.8GHz quad-cores Intel Xeon processors and 12Mb on-chip level 2 cache per processor. The chosen shared memory programming paradigm is OpenMp, which has become one of the most successful and widely used [24]. OpenMP is a directive-based, fork-join model for shared memory parallelism.

Care was taken when implementing Algorithm 7.3 to allocate the required memory for execution outside of the parallel region. When a thread allocates memory within a parallel region, a series of procedures are executed to ensure that no other thread will simultaneously access or alter this region of memory, and this entails serious overheads. A series of measures were also taken to diminish the number of accesses to the shared memory, this even entailed a change of data structure, but we will not go into these details for they extrapolate the

Algorithm 7.4: Parallel `edge_pushing_p`

Input: $x \in \mathbb{R}^n$, a tape $\mathcal{T}$, `nthreads`, `mythread`
Initialization $\texttt{enemyflag}_i = 0$, $\texttt{myflag}_i = 0$, for $i = 1 - n \dots \ell$,
`cur_pro` =0, tape $\hat{\mathcal{T}} = \emptyset$
for $i = \ell, \dots, 1$ **do**
  $\texttt{Pushing}_i$
  `Setting Flags`:
  if $\textit{myflag}_i = 0$ *and* $\textit{enemyflag}_i = 0$ **then**
    if ϕ_i *is a nonlinear function* **then**
      if *cur_pro* = *mythread* **then**
        $\texttt{myflag}_i = 1$;
      else
        $\texttt{enemyflag}_i = 1$
      `cur_pro` = `cur_pro`+1
      if *cur_pro* > *nthreads* **then**
        `cur_pro`=0
    end
  `Passing Flags`:
  if $\textit{myflag}_i = 1$ **then**
    $\texttt{Creating}_i$
    foreach $j \prec i$ **do**
      $\texttt{myflag}_j = 1$
    end
  if $\textit{enemyflag}_i = 1$ **then**
    foreach $j \prec i$ **do**
      $\texttt{enemyflag}_j = 1$
    end
  $\texttt{Adjoint}_i$
end
Output: A Graph structure of a Hessian matrix.

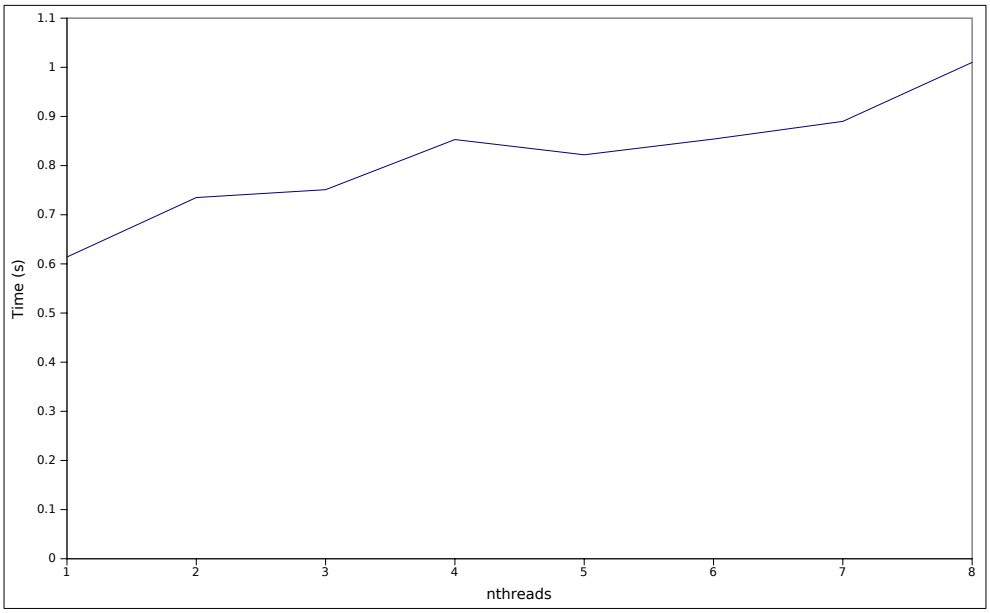


Figure 7.2: The runtime of `bdqtic` for Algorithm 7.3 with $n = 50'000$ and number of processors varying from 1 to 8.

scope of this dissertation.

The types of computational effort that can be boosted using threading in a multi-core shared memory computer, are ones that involve floating point arithmetic. While actions dominated by accessing and allocating memory have no speed-up in shared memory architecture, for threads share a large potion of memory, and cannot concurrently access and allocate it.

Our computational experiments show that the computational effort of the `edge_pushing_sp` algorithm is dominated by accessing and altering memory. What is more, the memory requirements for calculating a single H_{f_i} of (7.1) is of the same order of that of calculating the entire Hessian matrix. This is due to the overwriting scheme employed for `edge_pushing`, for the Hessians H_{f_i}, for $i = 1 \dots m$, tend to have very similar sparsity patterns, and edges allocated to calculate one H_{f_i} are re-used for the next one. Naturally threads cannot reuse each other's memory, thus more threads resulted in more memory allocation and a significant growth in allocating and deleting time. Large enough to compromise the entire execution time of the parallel `edge_pushing` strategy on many test functions. Look to Figure 7.2, which contains the results of executing Algorithm 7.3 on the function `bdqtic` with dimension $n = 50'000$ from the CUTE collection [5]. Similar results where obtained for all the functions tested in Section 5.3.

To illustrate when such a strategy would be worthwhile in a shared memory computer, we have elaborated an artificial function with a sparse Hessian such that the calculations in `edge_pushing_p` algorithm are dominated by floating point arithmetic, see the Heavy Arithmetic Function in Algorithm 7.5 and equations (7.3) and (7.4). In Figure 7.3 we have

plotted the runtime results of Algorithm 7.3 for the Heavy Arithmetic Function with `rep`, `nonl`, n and m equal to $20, 20, 50'000$ and $50'000$, respectively. There is a growing gain in using more threads on the Heavy Arithmetic Function up to four threads, after which it appears to stabilize around 45 seconds.

$$y_i = \prod_{j=1}^{\texttt{rep}} \prod_{k=i}^{i+\texttt{nonl}} \sin(x_k/j), \tag{7.3}$$

$$y = \sum_{i=1}^{m} y_i. \tag{7.4}$$

Algorithm 7.5: Heavy Arithmetic Function

Input: $x \in \mathbb{R}^n, \texttt{rep}, \texttt{nonl} \in \mathbb{N}$ and $m \in \mathbb{N}$
Initialization: $y = 0, y_i = 1$, for $i = 1 \dots m$.
for $i = 1 \dots m$ **do**
  for $j = 1 \dots$ *rep* **do**
    for $k = i \dots i +$ *nonl* **do**
      $y_i = \sin(x_k/j) * y_i;$
    end
  end
end
for $i = 1 \dots m$ **do**
  $y = y + y_i$
end
Output: y

In conclusion, using an implementation of `edge_pushing` aimed at sparse Hessians requires dynamic data structures, however dynamic allocation and shared memory programming do not mix. Thus our implementation of the Parallel `edge_pushing` strategy does not work well on shared memory computers due to dominance of memory related tasks. Nothing indicates that such a strategy would not be a success in a distributed system, thus this will be the focus of future work.

From a general outlook, a tape cutting strategy seems advantageous for it diminishes the problem size. The original tape has a size proportional to ℓ while, if m is large and the subgraphs headed by the f_i nodes are of similar sizes, Figure 7.1, then the severed pieces of the tape will be roughly proportional in size to $\ell/\texttt{nthreads}$. This being said, with taping and overwriting scheme of ADOL-C, the memory requirements of executing `edge_pushing` on the smaller tapes was the same as the memory requirements of the original tape.

As the technology of processing power advances, more and more will memory related operations become the bottleneck in modern software. This might well stimulate the production of hybrid computers with distributed sets of multi-core processors.

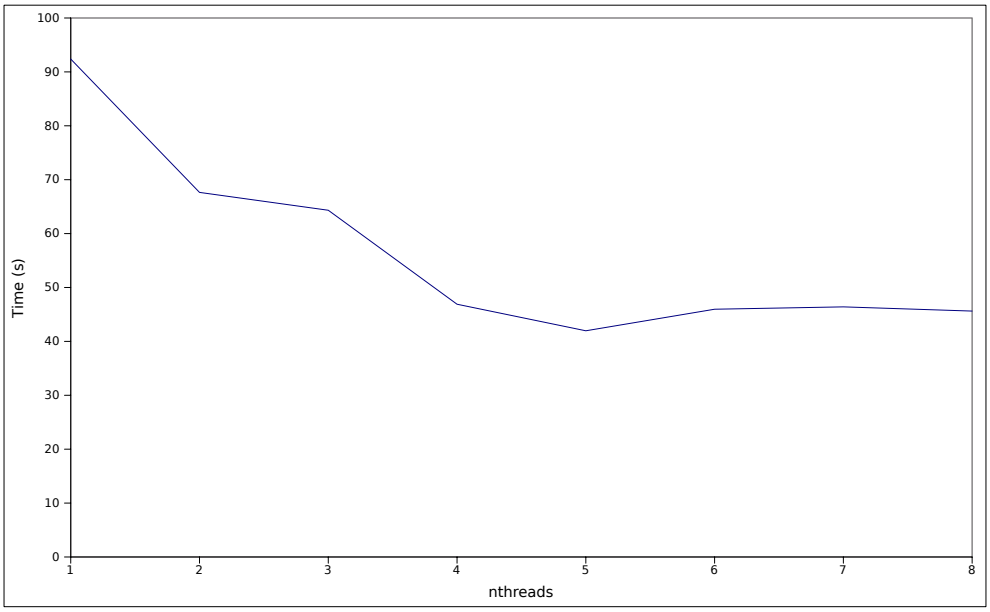


Figure 7.3: The runtime of Algorithm 7.3 for the function in Algorithm 7.5 with `rep`, `nonl` n and m equal to $20, 20, 50'000$ and $50'000$, respectively. And processors varying from 1 to 8.

Chapter 8

Conclusão

Resumindo as contribuições principais desta dissertação, desenvolvemos duas formas de caracterizar o problema de calcular a matriz hessiana. Usando estas caracterizações, criamos um novo algoritmo reverso para o cálculo da matriz hessiana: `edge_pushing`. Implementamos `edge_pushing`, o comparamos com o estado da arte e os resultados são promissores.

- **O grafo computacional e a matriz hessiana:** A primeira caracterização consistiu em descrever o cálculo das derivadas de segunda ordem como uma soma de pesos de caminhos especiais no grafo computacional. Esta visão expôs as simetrias inerentes ao problema e, com essa intuição, desenvolvemos um método reverso: `edge_pushing`. Este algoritmo tira proveito destas simetrias e acumula de forma eficiente todos os pesos destes caminhos especiais, calculando, assim, todas as derivadas parciais de segunda ordem.

- **A fórmula hessiana por transformações de estados:** Tomando a representação da função como uma sequência de transformações de estado [20], descrevemos a matriz hessiana como uma fórmula fechada, composta por somas e multiplicações das dervidas das transformações de estado, corolário (4.1). Com esta fórmula, elaboramos novas demonstrações de corretude de algoritmos existentes e desenvolvemos novos algoritmos. Fazendo um paralelo com o cálculo eficiente de polinômios por eninhamento, obtemos novamente o mesmo método reverso: `edge_pushing`.

- **O algoritmo `edge_pushing`:**

  O `edge_pushing` satisfaz a décima sexta regra de diferenciação automática do Griewank e Walther [20, p.240]:

  > *O cálculo do gradiente pelo método reverso torna o grafo computacional simétrico, uma propriedade que deve ser explorada e mantida ao acumular matrizes hessianas.*

  O `edge_pushing` realmente aproveita a esparsidade e a simetria da matriz hessiana. É um algoritmo de uma única fase, no sentido de que não há uma execução preparatória,

como nos métodos de coloração de grafo [14, 30], que precisam calcular o padrão de esparsidade da hessiana. Isso pode ser uma vantagem formidável quando se trata de funções cuja segunda derivada é descontínua, exigindo, portanto, o recálculo do padrão de esparsidade quando se muda de uma região contínua para outra. Por exemplo, $h(u) = (\max\{-u, 0\})^2$. Este tipo de função é usada como uma penalização diferenciável do eixo negativo.

Os testes computacionais do `edge_pushing`, comparandos com o método estrela e acíclico do ColPack [16], indicam que o `edge_pushing` possui uma execução mais veloz e é um algoritmo mais robusto.

8.1 Futuros desafios

- **Mais testes:** As funções de teste usadas foram tiradas da coleção CUTE [5], que, em sua maioria, são funções cujas hessianas possuem estruturas de esparsidades parecidas e, por isso, não são inteiramente adequadas para testar algoritmos que calculam matrizes hessianas. Logo, funções de teste cujas hessianas possuem estruturas de esparsidade mais diversificadas são importantes. Testes do `edge_pushing` inserido em um pacote de otimização não-linear também são necessários para apreciar o ganho em eficiência.

- **Uma implementação distribuída de `edge_pushing`:** Nossos testes e análises revelaram que `edge_pushing` não é adequado para adaptação e implementação em um sistema de memória compartilhada, devido ao fato de os cálculos do `edge_pushing` serem dominados por acessos e alterações de memória. Mas tudo indica que uma implementação em um sistema distribuído terá ganhos significativos.

- **Estender resultados para altas ordens:** Um interesse atual dos autores é desenvolver a teoria e possivelmente implementações de algoritmos reversos de diferenciação automática para o cálculo de derivadas de altas ordens. Apesar de que a dimensão de um tensor derivada aumenta com a ordem da diferenciação; a sua esparsidade e a sua simetria também aumentam. Um algoritmo reverso poderá aproveitar essa esparsidade e simetria e, por conseqüência, talvez tornar viável métodos de otimização de alta ordem.

Index

Bibliography

[1] Jason Abate, Christian Bischof, Lucas Roh, and Alan Carle. "Algorithms and design for a second-order automatic differentiation module". In: *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*. New York: ACM, 1997, 149–155 (electronic). DOI: 10.1145/258726.258770. URL: http://dx.doi.org/10.1145/258726.258770.

[2] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon. "Automatic differentiation of algorithms". In: *J. Comput. Appl. Math.* 124.1-2 (2000), pp. 171–190. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/S0377-0427(00)00422-2.

[3] Sanjukta Bhowmick and Paul D. Hovland. "A Polynomial-Time Algorithm for Detecting Directed Axial Symmetry in Hessian Computational Graphs". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 91–102. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_9.

[4] C. Bischof, G. Corliss, and A. Griewank. "Structured second-and higher-order derivatives through univariate Taylor series". In: *Optimization Methods and Software* 2.3 (1993), pp. 211–232. ISSN: 1055-6788.

[5] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. "CUTE: constrained and unconstrained testing environment". In: *ACM Trans. Math. Softw.* 21.1 (1995), pp. 123–160. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/200979.201043. URL: http://portal.acm.org/ft_gateway.cfm?id=201043&type=pdf&coll=Portal&dl=GUIDE&CFID=106302864&CFTOKEN=87967305.

[6] C. Büskens and H. Maurer. "Sensitivity analysis and real-time optimization of parametric nonlinear programming problems." In: *Online Optimization of Large Scale Systems: State of the Art*. Ed. by M. Grötschel, S. Krumke, and J. Rambau. Springer, 2001, pp. 3–16.

[7] R. H. Byrd, J. N., and R. A. Waltz. "KNITRO: An integrated package for nonlinear optimization". In: *Large Scale Nonlinear Optimization, 35–59, 2006*. Springer Verlag, 2006, pp. 35–59.

[8] Bruce Christianson. *Automatic Hessians by Reverse Accumulation*. Preprint MCS–P228–0491. 9700 S. Cass Ave., Argonne, IL 60439–4801: School of Information Sciences, University of Hertfordshire, 1990.

[9] Bruce Christianson. "Automatic Hessians by reverse accumulation". In: *IMA J. Numer. Anal.* 12.2 (1992), pp. 135–150. ISSN: 0272-4979. DOI: 10.1093/imanum/12.2.135. URL: http://dx.doi.org/10.1093/imanum/12.2.135.

[10] Anders Forsgren, Philip E. Gill, and Margaret H. Wright. "Interior methods for nonlinear optimization". In: *SIAM Review* 44 (2002), pp. 525–597.

[11] N. Ganesh and L. T. Biegler. "A reduced hessian strategy for sensitivity analysis of optimal flowsheets." In: 33 (1987), pp. 282–296.

[12] D. M. Gay. "More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability". In: *in Computational Differentiation: Applications, Techniques, and*. 1996, pp. 173–184.

[13] A. H. Gebremedhin, F. Manne, and A. Pothen. "What Color Is Your Jacobian? Graph Coloring for Computing Derivatives". In: *SIAM Rev.* 47.4 (2005), pp. 629–705. ISSN: 0036-1445. DOI: http://dx.doi.org/10.1137/S0036144504444711.

[14] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther. "Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation". In: *INFORMS J. on Computing* 21.2 (2009), pp. 209–223. ISSN: 1526-5528. DOI: http://dx.doi.org/10.1287/ijoc.1080.0286.

[15] Assefaw H. Gebremedhin, Arijit Tarafdar, Duc Nguyen, and Alex Pothen. *ColPack*. Sept. 2010. URL: http://www.cs.odu.edu/%7Ednguyen/dox/colpack/html/ (visited on 09/27/2010).

[16] Assefaw H. Gebremedhin, Arijit Tarafdar, and Alex Pothen. "COLPACK: A graph coloring package for supporting sparse derivative matrix computation". In preparation. 2008.

[17] R. M. Gower and M. P. Mello. *Hessian Matrices via Automatic Differentiation*. Tech. rep. Institute of Mathematics, Statistics and Scientific Computing, Unicamp, 2010.

[18] A. Griewank. *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*. Preprint MCS–P228–0491. 9700 S. Cass Ave., Argonne, IL 60439–4801: Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[19] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*. Tech. rep. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167. Institute of Scientific Computing, Technical University Dresden, 1999.

[20] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-451-6.

[21] Andreas Griewank. *Some Bounds on the Complexity of Gradients, Jacobians, and Hessians*. 1993.

[22] W. Hock and K. Schittkowski. "Test examples for nonlinear programming codes". In: *Journal of Optimization Theory and Applications* 30.1 (1980), pp. 127–129.

[23] R. H. F. Jackson and G. P. McCormick. "The polyadic structure of factorable function tensors with applications to high-order minimization techniques". In: *J. Optim. Theory Appl.* 51.1 (1986), pp. 63–94. ISSN: 0022-3239. DOI: http://dx.doi.org/10.1007/BF00938603.

[24] *OpenMP. A proposed industry standard API for shared memory programming.* 2010. URL: http://www.openmp.org/.

[25] R. Y. Rubinstein. "Sensitivity analysis and performance extrapolation for computer simulation models". In: *Oper. Res.* 37.1 (1989), pp. 72–81. ISSN: 0030-364X. DOI: http://dx.doi.org/10.1287/opre.37.1.72.

[26] James Stewart. *Multivariable Calculus*. Brooks Cole, 2007.

[27] Ph.L. Toint and A. Griewank. "On the unconstrained optimization of partially separable objective functions". In: ed. by M.J.D. Powell. Academic Press, London, 1982. Chap. Nonlinear Optimization, pp. 301–312.

[28] R. J. Vanderbei and D. F. Shanno. "An Interior-Point Algorithm For Nonconvex Nonlinear Programming". In: *Computational Optimization and Applications* 13 (1997), pp. 231–252.

[29] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program.* 106.1 (2006), pp. 25–57. ISSN: 0025-5610. DOI: http://dx.doi.org/10.1007/s10107-004-0559-y.

[30] A. Walther. "Computing sparse Hessians with automatic differentiation". In: *ACM Trans. Math. Softw.* 34.1 (2008), pp. 1–15. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/1322436.1322439.

[31] A. Walther and A. Griewank. *New Results on Program Reversals*. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 237–243. ISBN: 0-387-95305-1.

[32] R. E. Wengert. "A simple automatic derivative evaluation program". In: *Commun. ACM* 7.8 (1964), pp. 463–464. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/355586.364791.