The Determination of Structured Hessian Matrices via Automatic Differentiation

by

Samuel Embaye

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In using automatic differentiation (AD) for Hessian computation, efficiency can be achieved by exploiting the sparsity existing in the derivative matrix. However, in the case where the Hessian is dense, this cannot be done and the space requirements to compute the Hessian can become very large. But if the underlying function can be expressed in a structured form, a "deeper" sparsity can be exploited to minimize the space requirement. In this thesis, we provide a summary of automatic differentiation (AD) techniques, as applied to Jacobian and Hessian matrix determination, as well as the graph coloring techniques involved in exploiting their sparsity. We then discuss how structure in the underlying function can be used to greatly improve efficiency in gradient/Jacobian computation. We then propose structured methods for Hessian computation that substantially reduce the space required. Finally, we propose a method for Hessian computation where the structure of the function is not provided.

Acknowledgements

I want to thank my supervisor, Thomas Coleman, for his guidance and support throughout the master's program, and in completing this thesis. I also thank my readers Stephen Vavasis and George Labahn for their comments and insight. Finally, thank you to my parents for all of the support they have given me.

Table of Contents

List of Figures

1 Introduction

1.1 Overview

Scientific computing is an (almost) all-encompassing field in modern research, with a large variety of disciplines utilizing it for modelling and quantitative analysis of various phenomena. Efficiency in computation is the chief concern in these applications, and often the computation of derivatives is what requires the most time.

Automatic differentiation provides a very practical method to compute these derivatives. Unlike finite differencing methods, automatic differentiation does not incur truncation errors, and calculates derivatives to working precision. Unlike in symbolic differentiation, automatic differentiation only requires the computer code to evaluate a function, in order to determine its derivatives. It does not have to form potentially very complicated derivative function expressions in order to do so.

Automatic differentiation is becoming more and more widely utilized throughout scientific research, with fluid dynamics [30], mathematical biology [22], ship propulsion optimization [31], option pricing [27], and thermodynamics [28] being just a few of its many applications.

Often the Hessian matrix associated with a scalar-valued function is required in these applications and automatic differentiation performs efficiently in obtaining it in many situations. It does so by exploiting the sparsity inherent in the underlying function. However, for functions corresponding to a complicated computation where the Hessian is dense, sparse techniques are not very useful and the space required can become so large that using automatic differentiation becomes infeasible.

Thus if methods can be developed to use automatic differentiation in such a way as to mitigate the large space requirements for the Hessian matrix without significantly infringing on the efficiency in computing time, automatic differentiation becomes more generally applicable. This is the topic of this thesis.

1.2 Structure of Thesis

In this thesis we discuss structured methods to reduce space requirements in Hessian computation using automatic differentiation. We consider both the case where the function in question is provided in a structured form, and when it is not. **Chapter** 2 provides an overview of automatic differentiation, with a focus on a matrix representation of AD. **Chapter** 3 discusses how sparsity in the derivative matrix can be used to increase computational efficiency, and how graph theoretic concepts are invoked in doing so. **Chapter** 4 outlines the concept of structure, and how it can be used to improve efficiency in Jacobian and gradient calculations. In **Chapter** 5, we consider methods to exploit structure in Hessian computation while limiting the memory required, and numerical results are presented. In **Chapter** 6, we look at a structured method for the Hessian when the function is not provided in a structured form. Finally, **Chapter** 7 contains concluding remarks on the thesis and potential avenues for further research.

2 Automatic Differentiation

2.1 Overview

Automatic differentiation (AD) is a numerical method for determining derivative matrices of a real-valued function, given a computer program to evaluate the function. AD makes use of the fact that every computer code for evaluating a function uses a finite set of elementary operations and functions as defined by the programming language. This is done in such a way that the function computed by the computer program is simply a composition of these elementary functions and operations. The reason these elementary functions $(\sin, \cos, \exp, \log, \ldots)$ and operations $(+, -, \times, /, \ldots)$ are very useful in differentiation is that their derivatives and corresponding derivative operations, with respect to their inputs, are known and are easy to compute.

AD breaks the computer code for the function down into a partially ordered sequence of its elementary functions (we call this an evaluation procedure [26]). Then through repeated use of the chain rule, it calculates the function's derivatives accurately to working precision. Generally, for a function $F : \mathbb{R}^n \to \mathbb{R}^m$ the evaluation procedure is a three part process:

$$
\begin{array}{ll}
v_{i-n} = x_i & i = 1 \ldots n \\
\hline
v_i = \phi_i \left(v_j \right)_{\forall j < i} & i = 1 \ldots p \\
\hline
F_{m-i} = v_{p-i} & i = m-1 \ldots 0
\end{array}
$$

Figure 2.1: General Evaluation Procedure for AD

The input variables are converted to intermediate variables, a new intermediate variable is defined for every elementary operation ϕ_i, and finally the output variables for the function are extracted from the final intermediate variables. This evaluation procedure is used in two distinct ways to compute the derivative of the function, forward mode and reverse mode, which are discussed in **Section** 2.2 and 2.3, respectively.

2.2 Forward Mode AD

As mentioned before, automatic differentiation is essentially repeated application of the chain rule. Suppose we have a composite function of the following form:

$$f(x) = g(h(s(t(x)))) \tag{2.1}$$

then performing the chain rule on (2.1), with respect to *x* yields

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{ds}\frac{ds}{dt}\frac{dt}{dx} \tag{2.2}$$

Forward mode AD traverses the chain from right to left to obtain the function's derivative, as you would when evaluating the function. This allows for the function value and derivative to be computed concurrently [12]. We demonstrate this with an example in the succeeding section.

2.2.1 An Example of Forward Mode

Suppose we have a function $F : \mathbb{R}^3 \to \mathbb{R}^2$ defined as:

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin(x_3) \\ x_3 - \exp(x_1 x_2) \end{bmatrix} \tag{2.3}$$

and we would like to determine the Jacobian matrix $J \in \mathbb{R}^{m \times n}$, of F. AD determines the evaluation procedure of F as in Figure 2.1, by breaking down the function into a partially ordered set of elementary operations [26]. The evaluation procedure for (2.3) can be found in Figure 2.2.

$$
\begin{aligned}
&v_{-2} = x_1 \\
&v_{-1} = x_2 \\
&v_0 = x_3 \\
&v_1 = v_{-2} \cdot v_{-1} \\
&v_2 = \exp(v_1) \\
&v_3 = \sin(v_0) \\
&v_4 = v_1 + v_3 \\
&v_5 = v_0 - v_2 \\
&F_1 = v_4 \\
&F_2 = v_5
\end{aligned}
$$

Input variables

Intermediate variables

Output variables

Figure 2.2: Evaluation Procedure of (2.3)

Now suppose we want to compute the derivative of F with respect to x_2, in other words we want the derivatives $\frac{\partial F_1}{\partial x_2}$ and $\frac{\partial F_2}{\partial x_2}$. In forward mode AD, since we traverse the chain of elementary operations in the same direction as we do when evaluating the function, we can compute these derivative as we compute the function value. We define $\dot{v}_i = \frac{dv_i}{dx_2}$, so then $\dot{v}_{-1} = 1$ and $\dot{v}_{-2} = \dot{v}_0 = 0$ as they are independent variables. Putting this all together, we get the evaluation procedure for forward mode AD for the partial derivatives with respect to x_2.

$v_{-2} = x_1$	$\dot{v}_{-2} = 0$
$v_{-1} = x_2$	$\dot{v}_{-1} = 1$
$v_0 = x_3$	$\dot{v}_0 = 0$
$v_1 = v_{-2} \cdot v_{-1}$	$\dot{v}_1 = \dot{v}_{-2} \cdot v_{-1} + v_{-2} \cdot \dot{v}_{-1} = v_{-2}$
$v_2 = \exp(v_1)$	$\dot{v}_2 = \exp(v_1) \cdot \dot{v}_1 = \exp(v_1) \cdot v_{-2}$
$v_3 = \sin(v_0)$	$\dot{v}_3 = \cos(v_0) \cdot \dot{v}_0 = 0$
$v_4 = v_1 + v_3$	$\dot{v}_4 = \dot{v}_1 + \dot{v}_3 = v_{-2}$
$v_5 = v_0 - v_2$	$\dot{v}_5 = \dot{v}_0 - \dot{v}_2 = -\exp(v_1) \cdot v_{-2}$
$F_1 = v_4$	$\dot{F}_1 = \dot{v}_4$
$F_2 = v_5$	$\dot{F}_2 = \dot{v}_5$

Figure 2.3: An Evaluation Procedure for Forward Mode AD

We can see in Figure 2.3 that with one "sweep" through the function, we have recovered all the partial derivatives with respect to x_2; this corresponds to one column of the Jacobian. We can also see that the additional work required to get these partial derivatives is comparable to the work required to evaluate the function. So, if we define $\omega(F)$ as the work required to evaluate F, we can say that:

$$\omega\left(\frac{\partial F_1}{\partial x_2}, \frac{\partial F_2}{\partial x_2}\right) \sim \omega(F) \tag{2.4}$$

However, in order to fully compute the Jacobian J, we need these partial derivatives with respect to each one of the input variables. Unfortunately, as should be apparent from how we defined $\dot{v}_{-2}, \dot{v}_{-1}, \dot{v}_0$, for each input variable we must redefine these initial $\dot{v}_i$'s and compute a new sweep through the function F. Since F has *n* input variables, we come to the result that computing the Jacobian matrix using forward mode AD has cost:

$$\omega(J) \sim n \cdot \omega(F) \tag{2.5}$$

Additionally, since we are computing the derivative concurrently with evaluating the function, the space required to compute the derivative is just some multiple of the space required to evaluate the function. So if we define $\sigma(F)$ as the space required in evaluating the function F, we can say that:

$$\sigma(J) \sim \sigma(F) \tag{2.6}$$

2.3 Reverse Mode AD

Examining (2.1) again, we have a composite function $f(x) = g(h(s(t(x))))$, performing the chain rule with respect to x again yields:

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{ds}\frac{ds}{dt}\frac{dt}{dx} \tag{2.7}$$

Reverse mode AD traverses the chain from left to right. Since it begins with the output of the function, the function must be evaluated first to get the evaluation procedure. The procedure consisting of the intermediate values and operations done is saved to a "computational tape". Then the derivative is calculated by travelling backwards through the tape, until reaching the beginning of the tape where the derivative is recorded.

In order to record the derivatives as we travel back through the tape, we define a set of adjoint variables $\overline{v_i}$, corresponding to the existing intermediate variables v_i such that [12, 26]:

$$\overline{v_i} = \frac{dF}{dv_i} = \sum_{\forall k>i} \frac{dF}{dv_k}\frac{\partial \phi_k}{\partial v_i} = \sum_{\forall k>i} \overline{v_k}\frac{\partial \phi_k}{\partial v_i} \tag{2.8}$$

where $v_k = \phi_k\left(v_j\right)_{j<k}$. We initialize these adjoint variables to 0, then add to them incrementally as we traverse the tape backwards, until we reach the beginning of the tape and have fully accumulated the derivatives.

Generally, for a function $F : \mathbb{R}^n \to \mathbb{R}^m$, the evaluation procedure for performing reverse mode AD is as follows [26]:

$$
\begin{array}{lll}
\overline{v}_i = 0 & i = 1-n \ldots p & \left.\begin{array}{l}\\ \\ \\ \\ \end{array}\right\} \text{Function evaluation} \\
v_{i-n} = x_i & i = 1 \ldots n & \\
v_i = \phi_i\left(v_j\right)_{\forall j<i} & i = 1 \ldots p & \\
F_{m-i} = v_{p-i} & i = m-1 \ldots 0 & \\
\hline
\overline{v}_{p-i} = \overline{y}_{m-i} & i = 0 \ldots m-1 & \left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{Derivative evaluation} \\
\overline{v}_j = \overline{v}_j + \overline{v}_i \dfrac{\partial \phi_i}{\partial v_j} \text{ for } j < i & i = p \ldots 1 & \\
\overline{x}_i = \overline{v}_{i-n} & i = n \ldots 1 &
\end{array}
$$

Figure 2.4: General Evaluation Procedure for Reverse Mode AD

We will further illustrate reverse mode AD through an example in the next section.

2.3.1 An Example of Reverse Mode

Using the same example as in **Section** 2.2.1, we have a function $F : \mathbb{R}^3 \to \mathbb{R}^2$ such that:

$$
\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin(x_3) \\ x_3 - \exp(x_1 x_2) \end{bmatrix} \tag{2.9}
$$

Suppose we want to determine the Jacobian $J \in \mathbb{R}^{m \times n}$, of F. As was the case with forward mode AD, the function is broken down into a partially ordered set of elementary operations. However, in reverse mode, the function is evaluated in its entirety first, saving the ordered operations to a computational tape. Then the last *m* adjoint variables are initialized to user-specified values depending on with respect to which output the derivative is to be computed. Afterwards, the derivative is computed by sweeping through the tape in reverse.

For our example, we will compute the partial derivatives of F_1, which correspond to the first row of the Jacobian matrix. Following the general evaluation procedure for reverse mode in Figure 2.4, we first define the adjoint variables $\overline{v}_4 = 1$ and $\overline{v}_5 = 0$ (if we were getting the derivatives of F_2, we would define the adjoint variables as $\overline{v}_4 = 0$ and $\overline{v}_5 = 1$ instead). The evaluation procedure for the example is as follows:

$$\overline{v}_i = 0, \quad i = -2 \ldots 3$$
$$v_{-2} = x_1$$
$$v_{-1} = x_2$$
$$v_0 = x_3$$
$$v_1 = v_{-2} \cdot v_{-1}$$
$$v_2 = \exp(v_1)$$
$$v_3 = \sin(v_0)$$
$$v_4 = v_1 + v_3$$
$$v_5 = v_0 - v_2$$
$$F_1 = v_4$$
$$F_2 = v_5$$

$$\overline{v}_5 = 0$$
$$\overline{v}_4 = 1$$
$$\overline{v}_2 = \overline{v}_2 + \overline{v}_5 \cdot (-1) = 0$$
$$\overline{v}_0 = \overline{v}_0 + \overline{v}_5 \cdot (1) = 0$$
$$\overline{v}_1 = \overline{v}_1 + \overline{v}_4 \cdot (1) = \overline{v}_4 = 1$$
$$\overline{v}_3 = \overline{v}_3 + \overline{v}_4 \cdot (1) = \overline{v}_4 = 1$$
$$\overline{v}_0 = \overline{v}_0 + \overline{v}_3 \cdot \cos(v_0) = \cos(v_0)$$
$$\overline{v}_1 = \overline{v}_1 + \overline{v}_2 \cdot \exp(v_1) = \overline{v}_1 + \overline{v}_2 \cdot v_2 = 1$$
$$\overline{v}_{-1} = \overline{v}_{-1} + \overline{v}_1 \cdot v_{-2} = v_{-2}$$
$$\overline{v}_{-2} = \overline{v}_{-2} + \overline{v}_1 \cdot v_{-1} = v_{-1}$$

$$\overline{x}_3 = \overline{v}_0$$
$$\overline{x}_2 = \overline{v}_{-1}$$
$$\overline{x}_1 = \overline{v}_{-2}$$

Figure 2.5: An Evaluation Procedure for Reverse Mode AD

Although AD does all this numerically, analytically Figure 2.5 yields the results:

$$\frac{\partial F_1}{\partial x_1} = x_2, \quad \frac{\partial F_1}{\partial x_2} = x_1, \quad \frac{\partial F_1}{\partial x_3} = \cos(x_3) \tag{2.10}$$

By looking at (2.9), it can be easily verified that the results in (2.10) are correct. With one sweep through the function we have recovered all the partial derivatives of the first output variable; and looking at Figure 2.5 we see that computing these derivatives only costs a small multiple of the

cost to evaluate the function itself. Since the function F has *m* output variables (ie. the Jacobian has *m* rows), we can conclude that the work required to compute the Jacobian of a function F is:

$$\omega(J) \sim m \cdot \omega(F) \tag{2.11}$$

Additionally, since we have to save the entire computational tape from evaluating the function, in order to evaluate the derivative; reverse mode AD has space requirement:

$$\sigma(J) \sim \omega(F) \tag{2.12}$$

where typically, $\omega(F) >> \sigma(F)$. Reverse mode AD tends to have much higher storage demands than forward mode. However, its work's dependence on the number of output variables (rather than inputs), can be extremely useful. This is especially true for scalar-valued functions $(m=1)$, where the gradient would be computed.

2.4 Matrix Representation of AD

An alternative, but very useful way to look at AD is through the lens of matrix algebra. Suppose we have a program that evaluates the function $z = F(x),\ F : \mathbb{R}^n \to \mathbb{R}^m$. Evaluating the function using AD generates the partially ordered sequence of intermediate variables $(y_1, y_2, \ldots, y_p)$, where typically $p >> m, n$. Each intermediate variable y_i comes from an elementary operation on one $(\sin, \cos, \exp, \log, \ldots)$, or two $(+, -, \times, /, \ldots)$ of the previously computed intermediate and independent variables.

If we allow F^E to represent the "extended" version of the function F, we can express its decomposition into intermediate elementary functions as follows [15]:

$$\begin{aligned}
&\text{solve } y_1: && y_1 - F_1^E(x) = 0 \\
&\text{solve } y_2: && y_2 - F_2^E(x, y_1) = 0 \\
&\quad\vdots && \quad\vdots \\
&\text{solve } y_p: && y_p - F_p^E(x, y_1, y_2, \ldots, y_{p-1}) = 0 \\
&\text{solve output } z: && z - \overline{f}(x, y_1, y_2, \ldots, y_p) = 0
\end{aligned} \tag{2.13}$$

Differentiating the process in (2.13) with respect to all the independent and intermediate variables yields a $(p+m)\times(n+p)$ matrix C of partial derivatives, representing an extended Jacobian of F. If we order the independent and intermediate variables as $(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_p)$, then C can be written as:

$$C = \begin{pmatrix} F_x^E & F_y^E \\ \overline{f}_x & \overline{f}_y \end{pmatrix} \tag{2.14}$$

where:

$$F_x^E = \begin{pmatrix} \frac{\partial F_1^E}{\partial x_1} & \cdots & \frac{\partial F_1^E}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_p^E}{\partial x_1} & \cdots & \frac{\partial F_p^E}{\partial x_n} \end{pmatrix}, \quad F_y^E = \begin{pmatrix} \frac{\partial F_1^E}{\partial y_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \frac{\partial F_p^E}{\partial y_1} & \cdots & \frac{\partial F_p^E}{\partial y_p} \end{pmatrix},$$

$$\overline{f}_x = \begin{pmatrix} \frac{\partial \overline{f}_1}{\partial x_1} & \cdots & \frac{\partial \overline{f}_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \overline{f}_m}{\partial x_1} & \cdots & \frac{\partial \overline{f}_m}{\partial x_n} \end{pmatrix}, \quad \overline{f}_y = \begin{pmatrix} \frac{\partial \overline{f}_1}{\partial y_1} & \cdots & \frac{\partial \overline{f}_1}{\partial y_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial \overline{f}_m}{\partial y_1} & \cdots & \frac{\partial \overline{f}_m}{\partial y_p} \end{pmatrix} \tag{2.15}$$

C is a sparse matrix, since each elementary operation has at most two inputs, there are at most three non-zeroes in each row of C. It is also important to note that F_y^E is a lower triangular matrix, since each elementary function can only depend on the intermediate variables already calculated, and not the ones coming after it.

The Jacobian of F with respect to the independent variables *x*, can be recovered from (2.14) by a Schur complement computation:

$$J = \overline{f}_x - \overline{f}_y \left(F_y^E\right)^{-1} F_x^E \tag{2.16}$$

There are two principal ways to compute the Schur complement in (2.16) that differ in their order of operations, and they correspond to forward mode and reverse mode AD.

2.4.1 Forward Mode

The first method for computing (2.16) is:

$$J = \overline{f}_x - \overline{f}_y \left[\left(F_y^E \right)^{-1} F_x^E \right] \tag{2.17}$$

Computing $\left(F_y^E \right)^{-1} F_x^E$ requires the linear system $F_y^E W = F_x^E$ to be solved for W:

$$\begin{pmatrix} \frac{\partial F_1^E}{\partial y_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \frac{\partial F_p^E}{\partial y_1} & \cdots & \frac{\partial F_p^E}{\partial y_p} \end{pmatrix} \begin{pmatrix} W_1 \\ \vdots \\ W_p \end{pmatrix} = \begin{pmatrix} \frac{\partial F_1^E}{\partial x_1} & \cdots & \frac{\partial F_1^E}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_p^E}{\partial x_1} & \cdots & \frac{\partial F_p^E}{\partial x_n} \end{pmatrix} \tag{2.18}$$

Since F_y^E is lower-triangular, the W_i's can be determined from top to bottom. Since the matrix C is also created from top to bottom, as the function F is evaluated. We never need to store the entire matrix C. Thus the computational tape is only traversed in the forward direction, yielding forward mode AD, and so we have the space requirement in (2.6) [23].

We can also recover the same time complexity result for forward mode as in (2.5) [23], by examining (2.17). In forward mode AD, the Jacobian is computed column-wise, so we will determine the cost of computing the first column of the Jacobian.

Clearly, the first column of $\overline{f}_x$ is required, and looking at (2.15), this corresponds to the partial derivatives of $\overline{f}$ with respect to x_1. So this requires a forward sweep through the function, with cost proportional to that of evaluating the function, $\omega(F)$. The first column of $\overline{f}_y W$, where $W = \left(F_y^E \right)^{-1} F_x^E$, is also required. This corresponds to the first column of W, as well as the partial derivatives of $\overline{f}$ with respect to the intermediate variables y_i. This derivative information is recovered in the forward sweep, so there is no additional cost. Finally, from examining (2.15), we find that the first column of W requires the partial derivatives of F^E with respect to the intermediate variables y_i, and the partial derivatives of F^E with respect to x_1; both of which are also recovered in the forward sweep. Thus the cost of computing the first column of

the Jacobian, ignoring the cost of matrix multiplications and a sparse triangular system solve as this should be negligible in comparison to evaluating the function, is proportional to $\omega(F)$.

The Jacobian $J \in \mathbb{R}^{m \times n}$, has n columns, thus requiring n sweeps through the computational tape. We cannot do better than that here, because as we saw in **Section** 2.2.1 we can only compute derivatives with respect to one of the input variables per forward sweep; and we have n input variables. Thus we can conclude from a matrix approach just as we did in (2.5), that the cost of computing the Jacobian matrix of F is:

$$\omega(J) \sim n \cdot \omega(F) \tag{2.19}$$

2.4.2 Reverse Mode

The second method for computing (2.16) is:

$$J = \overline{f}_x - \left[\overline{f}_y \left(F_y^E \right)^{-1} \right] F_x^E \tag{2.20}$$

Computing $\overline{f}_y \left(F_y^E \right)^{-1}$ requires the linear system $\left(F_y^E \right)^T W^T = \overline{f}_y{}^T$ to be solved for W:

$$\begin{pmatrix} \frac{\partial F_1^E}{\partial y_1} & \cdots & \frac{\partial F_p^E}{\partial y_1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\partial F_p^E}{\partial y_p} \end{pmatrix} \begin{pmatrix} W_1^T & \cdots & W_p^T \end{pmatrix} = \begin{pmatrix} \frac{\partial \overline{f}_1}{\partial y_1} & \cdots & \frac{\partial \overline{f}_m}{\partial y_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial \overline{f}_1}{\partial y_p} & \cdots & \frac{\partial \overline{f}_m}{\partial y_p} \end{pmatrix} \tag{2.21}$$

This system is somewhat similar to (2.18), however since we are now dealing with $(F_y^E)^T$ rather than F_y^E, the left hand side of (2.21) is an *upper*-triangular matrix. This means it has to be solved bottom to top. However the matrix C is formed from top to bottom in evaluating F, so the entire matrix F_y^E (ie. the computational tape) has to be stored in order to compute (2.20). The computational tape is then traversed in the backwards direction, thus we have reverse mode AD.

Due to saving the computational tape in its entirety, we have the space complexity result $\sigma(J) \sim \omega(F)$ found in (2.12). We could also recover the time complexity result in the same way

we have done for forward mode in **Section** 2.4.1, yielding $\omega(J) \sim m \cdot \omega(F)$ as found in (2.11) [23].

2.4.3 Hessian Computation

Suppose we have a function $z = f(x),\ f : \mathbb{R}^n \to \mathbb{R}$. In order to determine the gradient, ∇f, which is the transpose of the Jacobian for a scalar-valued function, we would use reverse mode AD as $m = 1 \ll n$. Following (2.20), we can outline the procedure for determining the gradient :

$$\begin{aligned} &\text{Solve and differentiate (2.14) to obtain } C: && F^E(x, y) = 0 \\ &\text{Solve for } w: && (F_y^E)^T w + \overline{f}_y^T = 0 \\ &\text{Solve for } \nabla f: && (F_x^E)^T w + \overline{f}_x^T - \nabla f = 0 \end{aligned} \tag{2.22}$$

Then assuming all functions involved are twice differentiable, we differentiate (2.22) with respect to (x, y, w) to obtain an extended Hessian [14]:

$$H_E = \begin{pmatrix} F_x^E & F_y^E & 0 \\ \left(F_{yx}^E\right)^T w + \overline{f}_{yx} & \left(F_{yy}^E\right)^T w + \overline{f}_{yy} & \left(F_y^E\right)^T \\ \left(F_{xx}^E\right)^T w + \overline{f}_{xx} & \left(F_{xy}^E\right)^T w + \overline{f}_{xy} & \left(F_x^E\right)^T \end{pmatrix} \tag{2.23}$$

Then we obtain the Hessian matrix $H \equiv \nabla^2 f(x)$ by partitioning H_E:

$$H_E = \left(\begin{array}{c|cc} F_x^E & F_y^E & 0 \\ \left(F_{yx}^E\right)^T w + \overline{f}_{yx} & \left(F_{yy}^E\right)^T w + \overline{f}_{yy} & \left(F_y^E\right)^T \\ \hline \left(F_{xx}^E\right)^T w + \overline{f}_{xx} & \left(F_{xy}^E\right)^T w + \overline{f}_{xy} & \left(F_x^E\right)^T \end{array}\right) = \begin{pmatrix} A & L \\ B & M \end{pmatrix} \tag{2.24}$$

and then computing a Schur complement of H_E, as was done with C for Jacobian calculation [14].

$$H = B - ML^{-1}A \tag{2.25}$$

In order to compute the extended matrix H_E we have essentially differentiated the gradient routine, $\nabla f(x): \mathbb{R}^n \to \mathbb{R}^n$. Since the input and output variables are the same size there is no advantage in terms of work required to compute using either forward or reverse mode. There is no discernible space advantage either, since in the gradient routine we used reverse mode storing all of the computational tape, we already have it and would not incur significant additional space requirements by using it again. Thus we have the work complexity result for computing the Hessian *H* by AD of [23]:

$$\omega(H) \sim n \cdot \omega(f) \tag{2.26}$$

2.4.4 Derivative Matrix Products

A very useful property of automatic differentiation is that it can be used to compute Jacobian-matrix (and Hessian-matrix) products directly, without first explicitly computing the derivative matrix.

For a function $F: \mathbb{R}^n \to \mathbb{R}^m$, with Jacobian $J \in \mathbb{R}^{m \times n}$, suppose we have two matrices V, W such that $V \in \mathbb{R}^{n \times t_v}$ and $W \in \mathbb{R}^{m \times t_w}$. Then we can determine the product JV by forward mode AD in time proportional to $t_v \cdot \omega(F)$, and the product $W^T J$ by reverse mode AD in time proportional to $t_w \cdot \omega(F)$ [23, 24].

From (2.16), we have that:

$$JV = BV - M\left[L^{-1}AV\right] \tag{2.27}$$

$$W^T J = W^T B - \left[W^T M L^{-1}\right] A \tag{2.28}$$

Using the same argument as in **Section** 2.4.1, JV has t_V columns and $W^T J$ has t_W rows, thus requiring t_V forward sweeps, and t_W backward sweeps respectively. Each sweep costs $\omega(F)$, and thus we have the time complexity results [23, 24]:

$$\omega(JV) \sim t_V \cdot \omega(F) \tag{2.29}$$

$$\omega(W^T J) \sim t_W \cdot \omega(F) \tag{2.30}$$

Alternatively, the cost of computing the Jacobian and then afterwards the product JV, costs $n \cdot \omega(F)$ plus the cost of the matrix product [23]. So computing the product directly can result in significant time savings when the number of columns in V or W, is less than the number of columns, or rows respectively, in J.

We will see how useful this property can be in reducing the cost of AD when sparsity is accounted for, in the next chapter.

3 Sparsity and Coloring

3.1 Exploiting Sparsity

Typically in problems of large dimension, the Jacobian (or Hessian), exhibits some level of sparsity. Fortunately, AD can take advantage of this sparsity to enhance the efficiency of these derivative matrix computations [2, 3, 4]. As we've seen in Chapter 2, ignoring sparsity the (dense) Jacobian J of a function $F:\mathbb{R}^n \to \mathbb{R}^m$ can be computed using forward mode AD with cost proportional to $n \cdot \omega(F)$, or reverse mode AD with cost proportional to $m \cdot \omega(F)$ [23]. From a derivative matrix product standpoint as introduced in **Section** 2.4.4, these are equivalent to choosing [24]:

$$V = I_n \tag{3.1}$$

$$W = I_m \tag{3.2}$$

(where I_k represents the $k \times k$ identity matrix) and then computing the products JV and $W^T J$, for forward and reverse mode AD respectively.

However, if the Jacobian J is sparse, we can potentially choose thin matrices (ie. matrices with column dimension $\ll m,n$) V and/or W such that all the non-zeroes of J can be recovered from the products JV and/or $W^T J$. Thus computing the Jacobian can be done much more efficiently than if the identity matrices were used [8, 19]. We will consider a few often referenced examples to illustrate the potential for increased efficiency [12, 13].

Suppose we have a function $F:\mathbb{R}^n \to \mathbb{R}^n$, whose Jacobian is differentiable and has the structure (for $n = 5$):

$$J = \begin{pmatrix} J_{11} & & & & \\ J_{21} & J_{22} & & & \\ J_{31} & & J_{33} & & \\ J_{41} & & & J_{44} & \\ J_{51} & & & & J_{55} \end{pmatrix} \tag{3.3}$$

where J_{ij} represent the nonzero entries. If we were to ignore the sparsity in J and treat it as a dense matrix, computing it would require work proportional to $5 \cdot \omega(F)$ regardless of whether forward or reverse mode was used. However, if we select the thin matrix:

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.4}$$

Then the computation of JV by forward mode AD yields:

$$JV = \begin{pmatrix} J_{11} & 0 \\ J_{21} & J_{22} \\ J_{31} & J_{33} \\ J_{41} & J_{44} \\ J_{51} & J_{55} \end{pmatrix} \tag{3.5}$$

Comparing (3.5) to (3.3), all the nonzeros of J are contained within JV. Since V only has 2 columns, we have determined all the nonzeroes of J with work proportional to only $2 \cdot \omega(F)$ [24]. This is much more efficient than the $5 \cdot \omega(F)$ required if sparsity is ignored. The most astonishing result however, is that if we extend the problem to arbitrarily large size *n* (instead of 5), V still only needs to have 2 columns. Whereas dense forward mode AD now requires work proportional to $n \cdot \omega(F)$. When $n \gg 2$, we obtain significant cost reduction by exploiting the sparsity in J.

Let's now consider a similar example, but this time using reverse mode AD. Suppose the sparsity structure of J is:

$$J = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ & J_{22} & & & \\ & & J_{33} & & \\ & & & J_{44} & \\ & & & & J_{55} \end{pmatrix} \tag{3.6}$$

This time we select the thin matrix:

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.7}$$

and using reverse mode AD compute:

$$W^T J = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ 0 & J_{22} & J_{33} & J_{44} & J_{55} \end{pmatrix} \tag{3.8}$$

Once again, comparing (3.8) to (3.6), we have recovered all the nonzeroes of J in $W^T J$. Since W has 2 columns, computing the nonzeroes of J requires work proportional to $2 \cdot \omega(F)$ [24]. This is much more efficient than if sparsity is ignored where the cost is proportional to $5 \cdot \omega(F)$. If we extend the problem to size *n* with the same sparsity in the Jacobian, again the sparse method still costs $2 \cdot \omega(F)$, whereas using AD without considering sparsity now costs $n \cdot \omega(F)$. So for large *n*, exploiting sparsity can yield huge gains in efficiency of Jacobian calculation.

However, the potential savings in exploiting sparsity are not always as apparent as in the previous two examples. Now suppose the Jacobian has the following structure [32]:

$$J = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & & & \\ J_{31} & & J_{33} & & \\ J_{41} & & & J_{44} & \\ J_{51} & & & & J_{55} \end{pmatrix} \tag{3.9}$$

It turns out that because we have a fully dense row, forward mode AD cannot do better than $n \cdot \omega(F)$. Also, since we have a fully dense column, reverse mode AD cannot do better than $n \cdot \omega(F)$ either. However, if we use a combination of forward and reverse modes, we can still reduce the work required. Suppose we let:

$$V = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.10}$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.11}$$

Then forward mode AD yields:

$$JV = \begin{pmatrix} J_{11} \\ J_{21} \\ J_{31} \\ J_{41} \\ J_{51} \end{pmatrix} \tag{3.12}$$

and reverse mode AD yields:

$$W^T J = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ X & J_{22} & J_{33} & J_{44} & J_{55} \end{pmatrix} \tag{3.13}$$

Note that the entry X in $W^T J$ is nonzero, but not useful to us, so we discard it. Comparing (3.12), and (3.13) to (3.9), we have recovered all the nonzeroes of J, with cost proportional to $3 \cdot \omega(F)$, since V has one column and W has two [24]. This provides significant savings over the cost of only using one mode of AD, $n \cdot \omega(F)$. So generally, we may exploit sparsity by using a combination of both forward and reverse mode AD.

Then the question becomes, for a Jacobian J (or Hessian H) with general sparsity structure, how do we determine the most appropriate thin matrices V and/or W such that all the nonzeroes of J can be recovered from the products $JV, W^T J$? We will see in **Section** 3.2 and

Section 3.3 that this can be done in a number of ways by solving an associated graph coloring problem.

3.2 Coloring Jacobians

3.2.1 One-sided Methods

A one-sided method for automatic differentiation is one that utilizes just one of forward and reverse mode, but not both. Determining an appropriate matrix V (or W) to recover the nonzeroes of the Jacobian J is equivalent to finding a partition of the columns (or rows) of J.

A partition of the columns (or rows) of J is the division of the columns (or rows) into groups such that each column (or row) belongs to one and only one group. A column partition is said to be *consistent* with the direct determination of J if for each nonzero element J_{ij}, all the other columns that are in the same group as column *j* do not have a nonzero in row *i* (ie. each group consists of a structurally orthogonal set of columns). Similarly, a row partition is consistent with direct determination of J, if each group's set of rows is structurally orthogonal [11].

It is relatively easy to see how such a partition leads to a direct determination of J. We will demonstrate this for a consistent column partition. Suppose we have determined a column partition with groups $C_1, C_2, \ldots, C_p$, we define a set of column vectors v^i, $i = (1, \ldots, p)$ such that:

$$\left(v^i\right)_j = \begin{cases} 1 & \text{if } c_j \in C_i \\ 0 & \text{if } c_j \notin C_i \end{cases} \tag{3.14}$$

where c_j, $j = 1, \ldots, n$ is the n^{th} column of J. Then we concatenate the set of vectors into a matrix such that:

$$V = \left(v^1 \middle| v^2 \middle| \cdots \middle| v^p \right) \tag{3.15}$$

We now have our matrix V such that all the nonzeros of J can be recovered from the product JV, which is computed by forward mode AD. Since each group in the consistent column partition is structurally orthogonal, every entry in JV can only correspond to at most one nonzero entry in J. And since the partition covers all columns of J, we have directly determined all nonzero entries in J. The same argument can be made for a row partition consistent with direct determination.

We have shown that determining matrix V (or W) for derivative-matrix product computation is equivalent to finding a partition of the columns (or rows) of J consistent with direct determination of J. Each group in such a partition corresponds to a column in V (or W), and in order to be most efficient we need the matrix V (or W) to be as thin as possible. This means we can reformulate the problem presented at the end of **Section** 3.1, to find the thinnest matrix V (or W) such that all the nonzeros of J can be recovered, as a partitioning problem. The partitioning problem is to find a consistent column (or row) partition of J, requiring the least number of groups [11].

It turns out that this partitioning problem is equivalent to a certain graph-coloring problem. If we have a graph $G = (V, E)$, a coloring of the graph, $\phi : V \to \{1, 2, \ldots, p\}$, is a labelling of the graph's vertices such that no two vertices sharing an edge have the same color. The chromatic number of the graph, $\chi(G)$, is the minimum number of colors required to color the graph. Determining the optimal coloring of an arbitrary graph, one with $\chi(G)$ colors, is an NP-complete problem [29], so typically a heuristic algorithm is used to determine a near-optimal graph coloring.

If a consistent column partition is desired, the equivalent graph coloring problem is to find an optimal coloring of the column intersection graph of J, $G_U(J) = (V, E)$ where the vertex set $V = \{c_1, c_2, \ldots, c_n\}$ corresponds to the columns of J, and there exists an edge (c_i, c_j) if and only if both the i^{th} and j^{th} columns of J have a nonzero in the same row position [11, 20].

We will illustrate this equivalency by looking back at the example in (3.3). The column intersection graph associated with J in (3.3) is:

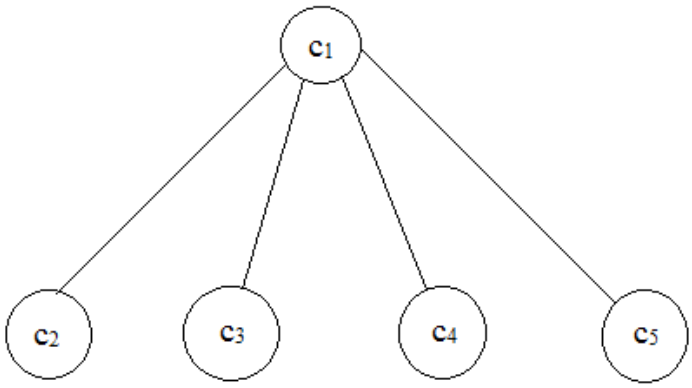

Figure 3.1: Column Intersection Graph of (3.3)

This is a relatively simple graph, so we can see that the optimal coloring would be:

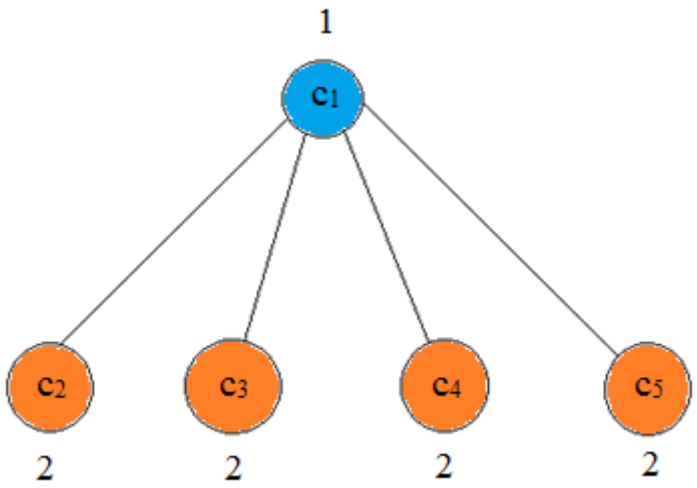

Figure 3.2: Optimal Coloring of Column Intersection Graph

If we allow each color to represent a group, then by (3.14) and (3.15), we have recovered the matrix V in (3.4).

Analogously, the consistent row partition corresponds to coloring the Jacobian's associated row intersection graph $G_U(J) = (V, E)$. Where the vertex set $V = \{r_1, r_2, \ldots, r_m\}$ corresponds to the rows of J, and there exists an edge (r_i, r_j) if and only if both the i^{th} and j^{th} rows of J have a nonzero in the same column position. We will not demonstrate this with an example like we did for the column case, as the resulting graph and coloring are identical to those found in Figure 3.1 and Figure 3.2.

There are several algorithms that can be used to determine a near optimal coloring of the graph, such that the number of colors used, $p \approx \chi(J)$, where $\chi(J)$ is the chromatic number of the column (or row) intersection graph of J [8, 11, 20]. $\chi(J)$ gives us a lower bound on the number of colors required and it is always true that $\chi(J) \leq n$ (or m), since we can always assign a different color to each vertex and have a permissible coloring. This means that by exploiting sparsity, we can compute the nonzeroes of J using a one-sided method with work:

$$\omega(J) \sim \chi(J) \cdot \omega(F) \tag{3.16}$$

and this will always cost no more than it would have if we had ignored sparsity [11].

3.2.2 Bicoloring Methods

As we have seen in examining the sparsity structure of J in (3.9), a one-sided method cannot always take advantage of the sparsity in J . We claimed that this was the case because (3.9) contained both a dense column and row. Now we will demonstrate why this is the case by looking at the column intersection graph of (3.9) (the row intersection graph is identical):

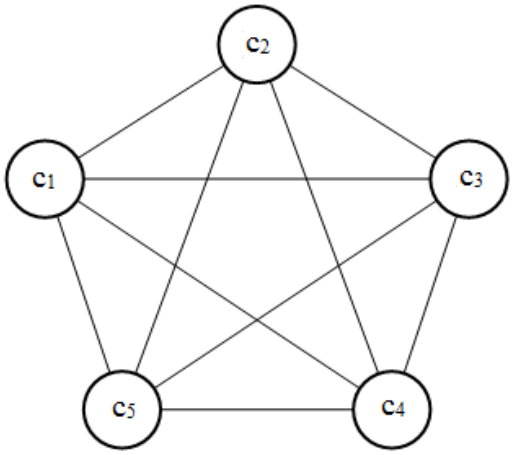

Figure 3.3: Column Intersection Graph of (3.9)

It is apparent from Figure 3.3 that a coloring of this graph must have n (or m) colors, and so there is no improvement in time complexity compared to if we had ignored the sparsity.

However, as we saw in (3.10)-(3.13) the computation of the nonzeroes of J in (3.9) can be done much more efficiently if we use a combination of forward and reverse mode AD, by

what is called a bicoloring method [13]. A similar method for determining a sparse Jacobian by both columns and rows is given by Hossain and Steihaug [33]. In order to do this, we need a *bipartition* of the Jacobian J. A bipartition of J is a pair (G_R, G_C) where G_R is a row partition of a subset of the rows of J, and G_C is a column partition of a subset of the columns of J. A bipartition (G_R, G_C) is consistent with *direct* determination if for every nonzero entry in J, a_{ij}, either column *j* is in a group of G_C which has no other column having a nonzero in row *i*; or row *i* is in a group of G_R which has no other rows having a nonzero in column *j* [13].

The analogous bipartitioning problem is to find a consistent bipartition such that $|G_R| + |G_C|$ is as small as possible, where $|G_C|$ represents the number of groups in G_C and $|G_R|$ represents the number of groups in G_R [13]. As shown in the one-sided case, given a bipartition consistent with direct determination, it is simple to construct matrices $V \in \mathbb{R}^{n \times |G_C|}$ and $W \in \mathbb{R}^{m \times |G_R|}$ such that J can be directly determined from the products JV and $W^T J$.

The bipartitioning problem can be interpreted as a restricted graph coloring problem. This time we need a bipartite graph associated with the Jacobian. Given a matrix $J \in \mathbb{R}^{m \times n}$, we define a bipartite graph $G_b(J) = ([V_1, V_2], E)$ where the vertex sets correspond to the set of columns of J, $V_1 = \{c_1, c_2, \ldots, c_n\}$, and the set of rows of J, $V_2 = \{r_1, r_2, \ldots, r_m\}$; edges exist only between the two vertex sets such that there exists an edge (c_j, r_i) if and only if the J_{ij} entry of J is nonzero. The associated bipartite graph of (3.9) is:

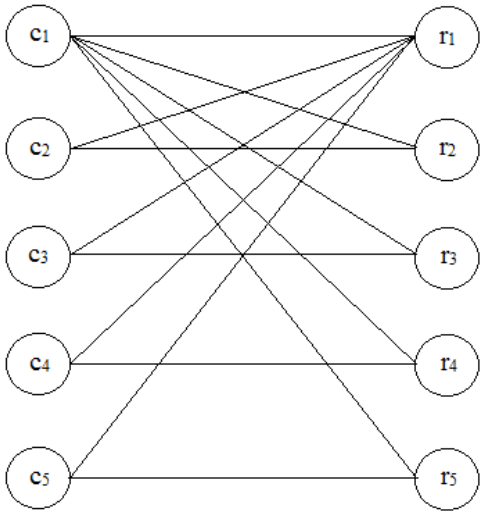

Figure 3.4: Associated Bipartite Graph of (3.9)

We then need to find a bipartite path *p*-coloring of $G_b(J)$. A bipartite path *p*-coloring is a coloring using *p* colors, $\phi:[V_1,V_2]\to\{0,1,\ldots,p\}$, with the additional properties that every path of at least three edges must use at least three colors, and color 0 corresponds to a lack of an actual color assignment. The minimum number of colors required for a path *p*-coloring of graph $G_b(J)$, is $\chi_p(G_b(J))$, which we will call the direct bichromatic number [13]. An optimal path 3-coloring for the bipartite graph in Figure 3.4 is:

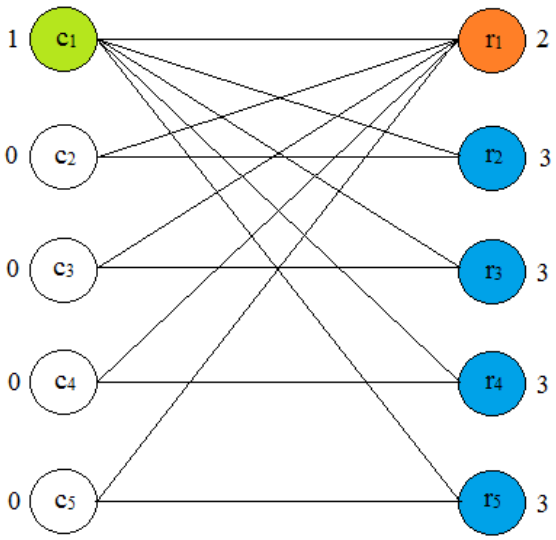

Figure 3.5: Path *p*-coloring of Bipartite Graph

where the white nodes correspond to the 0-color. Since we do not consider the 0-color a true color, this path 3-coloring corresponds to a bipartitioning (G_R, G_C) where $G_R = \{[1], [2,3,4,5]\}$ and $G_C = \{[1]\}$. This bipartitioning in turn corresponds to matrices V, W where:

$$V = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{ and } W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.17}$$

as in (3.10), and (3.11).

Generally, finding an optimal path *p*-coloring of $G_b(J)$ cannot be done efficiently. However, there exists heuristics [1, 13, 20] that yield a near optimal path *p*-coloring such that $p \approx \chi_p(J)$, where $\chi_p(J)$ is the direct bichromatic number for the graph $G_b(J)$. This leads to the time complexity result for Jacobian computation using the direct bicoloring method of:

$$\omega(J) \sim \chi_p(J) \cdot \omega(F) \tag{3.18}$$

It is important to note that the direct bicoloring method will always perform at least as well as the direct one-sided methods [13]:

$$\chi_p(J) \leq \chi(J) \tag{3.19}$$

This is because the path *p*-coloring problem for the graph $G_b(J)$, can be made equivalent to coloring the column intersection graph $G_U(J)$, by simply requiring that the path coloring $\phi : [V_1, V_2] \rightarrow \{0,1,\ldots, p\}$, maps the vertex set corresponding to the rows to the 0-color (ie. $\phi : [V_2] \rightarrow \{0\}$) and the column set to the positive colors (ie. $\phi : [V_1] \rightarrow \{1,2,\ldots, p\}$); and vice versa for the row intersection graph.

We can further improve on the time complexity bound found in (3.18) if we relax our requirement that all the nonzeroes of the Jacobian are directly determined, and instead allow for them to be determined by a substitution method.

This substitution method requires a new bipartition, one that is consistent with determination by substitution. A bipartition (G_R, G_C) is consistent with determination by *substitution* if there exists an ordering π on elements a_{ij} such that for every nonzero a_{ij} of *A*, either column *j* is in a group where all nonzeros in row *i* from other columns in the group are ordered lower than a_{ij}; or row *i* is in a group where all the nonzeros in column *j* from other rows in the group are ordered lower than a_{ij}. The bipartitioning problem is to find such a consistent bipartitioning where $|G_R| + |G_C|$ is minimized [13].

Once again, the bipartitioning problem can be interpreted as a graph coloring problem. As with the direct bicoloring method, the graph in question is the bipartite graph associated with the Jacobian, $G_b(J) = ([V_1, V_2], E)$, where the vertex sets correspond to the columns and rows, and there exists an edge (c_j, r_i) if and only if the J_{ij} entry of J is nonzero. For the substitution method, we require a bipartite cyclic *p*-coloring of $G_b(J)$. A bipartite cyclic *p*-coloring is a coloring using *p* colors, $\phi : [V_1, V_2] \rightarrow \{0, 1, \ldots, p\}$, with the additional properties that every cycle must use at least three colors, and color 0 corresponds to a lack of an actual color assignment [13, 20].

The minimum number of colors required for a valid bipartite cyclic *p*-coloring is $\chi_c(J)$, which we will call the substitution bichromatic number. This leads to the time complexity bound for the substitution bicoloring method of AD:

$$\omega(J) \sim \chi_c(J) \cdot \omega(F) \tag{3.20}$$

It should be relatively clear that the substitution method will always perform at least as well as the direct method. This is because the requirements for a cyclic *p*-coloring are not as strict as those for a path *p*-coloring. A cyclic *p*-coloring requires that all cycles use at least 3 colors, but every cycle is a path of at least 3 edges (cycles with 1 or 2 edges are impermissible). This means that every path *p*-coloring is a cyclic *p*-coloring, but the converse is clearly not true. Thus leading to the result regarding the bichromatic numbers that:

$$\chi_c(J) \leq \chi_p(J) \tag{3.21}$$

Combining the inequalities in (3.19) and (3.21) we have that:

$$\chi_c(J) \leq \chi_p(J) \leq \chi(J) \tag{3.22}$$

This shows that the substitution bicoloring AD method performs better than the direct bicoloring AD method, which in turn performs better than both one-sided AD methods.

Substitution methods force another thing to be considered, the propagation of error through substitutions. However, it has been showed that with certain conditions this can be well mitigated [13].

3.3 Coloring Hessians

The Hessian matrix of a function $f : \mathbb{R}^n \to \mathbb{R}$ is a symmetric matrix $\nabla^2 f \in \mathbb{R}^{n \times n}$. The chief difference when dealing with the sparsity of a Hessian, versus that of a Jacobian, is this symmetry which typically does not exist in a Jacobian. This symmetry opens new avenues for exploiting sparsity [6, 7, 10, 32]. Naturally, if we choose to ignore the symmetry in $\nabla^2 f$, all of the methods discussed in **Section** 3.2 are applicable. However, we will demonstrate how exploiting the existing symmetry in $\nabla^2 f$ can lead to further gains in efficiency.

We can demonstrate this rather simply by looking back at the sparsity structure in (3.9), but now we consider it as a symmetric Hessian's structure [32]:

$$\nabla^2 f = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} \\ h_{21} & h_{22} & & & \\ h_{31} & & h_{33} & & \\ h_{41} & & & h_{44} & \\ h_{51} & & & & h_{55} \end{pmatrix} \tag{3.23}$$

If the symmetry is ignored, we saw in **Section** 3.2 that one sided methods yielded a partition consistent with direct determination containing 5 groups, whereas the direct bicoloring method yielded a bipartition consistent with direct determination containing only 3 groups. However, if we exploit the symmetry in (3.23), we can form a valid partition consisting of only 2 groups.

Suppose we choose to partition the columns of $\nabla^2 f$ such that we have the two groups: $C_1 = \{1\}$, and $C_2 = \{2,3,4,5\}$. This partitioning leads to the seed matrix:

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.24}$$

and the Hessian-matrix product computed by AD:

$$HV = \begin{pmatrix} h_{11} & 0 \\ h_{21} & h_{22} \\ h_{31} & h_{33} \\ h_{41} & h_{44} \\ h_{51} & h_{55} \end{pmatrix} \tag{3.25}$$

Now, it may not appear so initially, but because the Hessian $\nabla^2 f$ is symmetric, we have directly recovered all the nonzeroes of $\nabla^2 f$ in the product HV. Since $\nabla^2 f$ is symmetric, we have the property that $h_{ij} = h_{ji}, \forall i, j$; so in determining the first column of $\nabla^2 f$ (in the first column of HV) we have determined the first row of $\nabla^2 f$ as well. Then we get the remaining nonzeroes on the diagonal from the second column of HV. Thus we have recovered the nonzeroes of the Hessian with only 2 groups, better than both methods ignoring symmetry.

Generally, exploiting the sparsity and symmetry of the Hessian to recover its nonzeroes requires a symmetrically consistent partition of the columns of $\nabla^2 f$. A partition is symmetrically consistent with direct determination of $\nabla^2 f$ if whenever h_{ij} is a nonzero element of $\nabla^2 f$ then the group containing column *j* has no other column with a nonzero in row *i*; *or* the group with column *i* has no other column with a nonzero in row *j* [10]. Note that the definition of a symmetrically consistent partition is not as strict as that of a consistent partition. The partitioning problem is then to find a symmetrically consistent partition of the columns of $\nabla^2 f$ containing as few groups as possible.

The graph theoretic interpretation of this partitioning problem involves the adjacency graph associated with the Hessian, $G_S(H)$. The adjacency graph is defined such that the vertex set $V = \{h_1, h_2, \ldots, h_n\}$ corresponds to the columns of the Hessian, and there exists edge (h_i, h_j) if and only if $i \neq j$ and $h_{ij} \neq 0$. We then require a symmetric *p*-coloring of the graph $G_S(H)$. A symmetric *p*-coloring is a coloring using *p* colors, $\phi : [V] \rightarrow \{1, \ldots, p\}$, with the additional property that every path of length 3 uses 3 colors [10]. Note that this is very similar to the path *p*-coloring definition for the unsymmetric direct bicoloring method [13], but we do not require the graph to be bipartite here. The minimum number of colours required for a valid symmetric *p*-coloring is $\chi_\sigma(H)$, which we will call the symmetric chromatic number. Thus we have the time complexity result for symmetric direct determination of the Hessian using AD of:

$$\omega(\nabla^2 f) \sim \chi_\sigma(\nabla^2 f) \cdot \omega(F) \tag{3.26}$$

Let us consider an example where the Hessian has tridiagonal sparsity structure:

$$\nabla^2 f = \begin{pmatrix} h_{11} & h_{12} & & & \\ h_{21} & h_{22} & h_{23} & & \\ & h_{32} & h_{33} & h_{34} & \\ & & h_{43} & h_{44} & h_{45} \\ & & & h_{54} & h_{55} \end{pmatrix} \tag{3.27}$$

This Hessian has the corresponding adjacency graph:

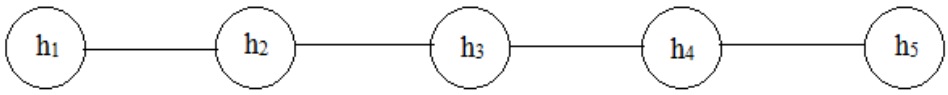

Figure 3.6: Associated Adjacency Graph for (3.27)

Since this is a very simple graph, we can easily determine a minimum symmetric 3-coloring:

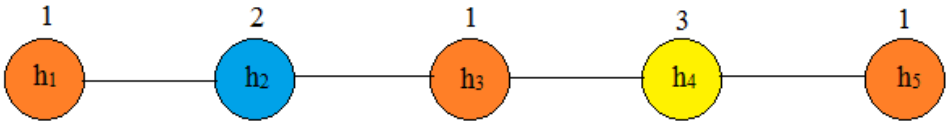

Figure 3.7: Symmetric *p*-coloring of Adjacency Graph

We can show how this coloring leads to direct determination of the Hessian by coloring its entries directly determined with their corresponding color:

$$\nabla^2 f = \begin{pmatrix} 1 & 2 & & & & \\ 2 & 2 & 2 & & & \\ & 2 & 1 & 3 & & \\ & & 3 & 3 & 3 & \\ & & & 3 & 1 & \end{pmatrix}$$

Figure 3.8: Sparsity Structure with Associated Colors

Thus we can compute the nonzeroes of the Hessian directly from a partition consisting of only 3 groups.

As with the unsymmetric bicoloring methods, we can further improve the efficiency of the Hessian calculation by allowing for entries to be determined by substitution. In order to do this, we need a partition consistent with determination by substitution. Generally, a partition of the columns of the Hessian induces a substitution method if there exists an ordering of the nonzeroes such that whenever h_{ij} is a nonzero element of $\nabla^2 f$, the group containing column j can only have another column with a nonzero in row i if that nonzero is previously ordered; or the group with column i only has another column with a nonzero in row j if that nonzero is previously ordered [6].

The graph theoretic approach to finding a smallest partition consistent with substitution involves finding a cyclic *p*-coloring of the associated adjacency graph, $G_S(H)$; which we defined in **Section** 3.2.2 for a bipartite graph. However in this context, we define the minimum number

of colors required for a valid cyclic *p*-coloring as the cyclic chromatic number, $\chi_0(H)$ [6]. Similar to how it was done in **Section** 3.2.2, it can be demonstrated that the substitution method will always perform at least as well as the direct method since:

$$\chi_o(H) \le \chi_\sigma(H) \tag{3.28}$$

We can demonstrate this by looking back at the sparsity structure in (3.27). A minimum cyclic 2-coloring for the associated adjacency graph is given in Figure 3.9.

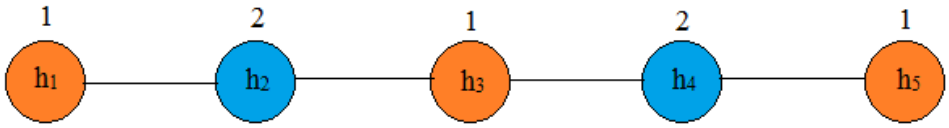

Figure 3.9: Cyclic *p*-coloring of Adjacency Graph

This corresponds to the seed matrix and Hessian-matrix product:

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad HV = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} + h_{23} & h_{22} \\ h_{33} & h_{32} + h_{34} \\ h_{43} + h_{45} & h_{44} \\ h_{55} & h_{54} \end{pmatrix} \tag{3.29}$$

Then if we choose to order the nonzeroes using ordering S, where:

$$S = \{h_{11}, h_{22}, h_{33}, h_{44}, h_{55}, h_{12}, h_{54}, h_{21}, h_{45}, h_{23}, h_{43}, h_{32}, h_{34}\} \tag{3.30}$$

Then we can determine all the nonzeroes of $\nabla^2 f$ by substitution using a partition consisting of only two groups; compared to needing three groups when doing so directly.

Generally, there exist many schemes for determining a near-optimal cyclic *p*-coloring, and corresponding permissible orderings of the nonzeroes [6, 21].

4 Structure

4.1 General Framework

We have examined methods to exploit sparsity in Jacobian and Hessian computations, allowing considerable speedup in AD implementation. But oftentimes we may deal with a function resulting in a large computation that does not have sparsity in its Jacobian or Hessian. Thus, when AD is applied in a straightforward manner, it becomes prohibitively expensive, from a time and/or space standpoint [9]. However, we can take advantage of the underlying structure in the function that often exists. This allows us to get to sparsity that can be exploited, and greatly decrease the cost of Jacobian computation.

For example, suppose we have a composite function $z = F(x)$, with the following structure which is very common for large-scale problems [15]:

$$F(x) = \bar{F}(y) \tag{4.1}$$

where y is the solution to a large, sparse, positive definite system:

$$A(x)y = \tilde{F}(x) \tag{4.2}$$

Through use of the chain rule, we can determine the Jacobian J, of $F(x)$:

$$J = \frac{dF}{dx} = \frac{d\bar{F}(y)}{dx} = \frac{d\bar{F}}{dy} \cdot \frac{dy}{dx} = \bar{J} \cdot \frac{dy}{dx} \tag{4.3}$$

where $\bar{J}$ is the Jacobian of $\bar{F}$. If we differentiate (4.2) with respect to x, we can determine $\frac{dy}{dx}$:

$$\begin{aligned} \frac{d}{dx}\left(A(x)y\right) &= \frac{d}{dx}\left(\tilde{F}(x)\right) \\ A_x y + A\frac{dy}{dx} &= \tilde{J} \\ \frac{dy}{dx} &= A^{-1}\left(\tilde{J} - A_x y\right) \end{aligned} \tag{4.4}$$

where $\tilde{J}$ is the Jacobian of $\tilde{F}$, and $A_x y$ is the Jacobian of the mapping $A(x)y$. Thus from (4.3) and (4.4) we have the expression for the Jacobian:

$$J = \bar{J}A^{-1}\left(\tilde{J} - A_x y\right) \tag{4.5}$$

Because of the application of A^{-1}, the Jacobian J will almost always be dense, even when $\bar{J}$, $\tilde{J}$, and $A_x y$ are sparse matrices. Thus, if we apply the sparse AD techniques discussed in Chapter 3 to determine the Jacobian, we make no gains in efficiency. However, if we exploit the structure in this function, we can find underlying sparsity in the function at a deeper level. Consider the following procedure to evaluate the function $z = F(x)$ at a given x [15]:

$$\begin{aligned} &\text{Solve for } y_1: \quad y_1 - \tilde{F}(x) = 0 \\ &\text{Solve for } y_2: \quad A y_2 - y_1 = 0 \\ &\text{Solve for } z: \quad z - \bar{F}(y_2) = 0 \end{aligned} \tag{4.6}$$

If we consider this procedure as an extended function F_E with respect to $\left(x, y_1, y_2\right)$ where:

$$F_E\left(x, y_1, y_2\right) = \begin{pmatrix} y_1 - \tilde{F}(x) \\ A(x)y_2 - y_1 \\ -\bar{F}(y_2) \end{pmatrix} \tag{4.7}$$

Then differentiating (4.7) with respect to $\left(x, y_1, y_2\right)$ yields the extended Jacobian:

$$J_E = \begin{pmatrix} -\tilde{J} & I & 0 \\ A_x y_2 & -I & A \\ 0 & 0 & -\bar{J} \end{pmatrix} \tag{4.8}$$

Clearly J_E is sparse, and thus we can use sparse AD techniques, such as the bicoloring method of **Section** 3.2.2, to compute it. This would require work:

$$\omega(J_E) \sim \chi_b(J_E) \cdot \omega(F_E) = \chi_b(J_E) \cdot \omega(F) \tag{4.9}$$

since the extended function F_E is just an alternative representation of the function F [13]. Typically, the bichromatic number $\chi_b(J_E) \ll m, n$. To what extent depends on how sparse J_E is,

so computing J_E requires much less work than computing the dense J. Then the Jacobian J can be recovered from J_E by a Schur complement computation [15] such that:

$$J = \bar{J}A^{-1}\left(\tilde{J} - A_x y\right) \tag{4.10}$$

The algebraic work required for the Schur complement is typically small compared to that of determining J_E, and so using structure to allow for sparsity exploitation can yield large gains in efficiency.

There are many well-known problems that exhibit structure that can be utilized, with partially-separable functions, dynamic systems and composite functions being a few. Restricting ourselves to scalar-valued functions, we represent a general structured computation of $z = f(x): \mathbb{R}^n \to \mathbb{R}$ as follows [14]:

$$\begin{aligned}
&\text{Solve for } y_1: \quad M_1 y_1 - F_1\left(x\right) = 0 \\
&\text{Solve for } y_2: \quad M_2 y_2 - F_2\left(x, y_1\right) = 0 \\
&\quad \vdots \\
&\text{Solve for } y_p: \quad M_p y_p - F_p\left(x, y_1, y_2, \ldots, y_{p-1}\right) = 0 \\
&\text{Solve for output } z: \quad z - \bar{f}\left(x, y_1, y_2, \ldots, y_p\right) = 0
\end{aligned}$$

Figure 4.1: General Structured Computation

where the F_i's $(i = 1{:}\, p)$ and $\bar{f}$ are intermediate functions, and the intermediate variables are $y_i \in \mathbb{R}^{n_i}$, $i = 1{:}\, p$. In Figure 4.1, each M_i is nonsingular and its order is equal to the length of vector y_i. The corresponding extended Jacobian can be written as:

$$J_E = \begin{pmatrix}
J_x^1 & -M_1 & & & \\
J_x^2 & J_{y_1}^2 & -M_2 & & \\
\vdots & \vdots & \vdots & \ddots & \\
J_x^p & J_{y_1}^p & \vdots & J_{y_{p-1}}^p & -M_p \\
\nabla_x \bar{f}^T & \nabla_{y_1} \bar{f}^T & \cdots & \cdots & \nabla_{y_p} \bar{f}^T
\end{pmatrix} \tag{4.11}$$

If we partition J_E as:

$$J_E = \left(\begin{array}{c|cccc} J_x^1 & -M_1 & & & \\ J_x^2 & J_{y_1}^2 & -M_2 & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & \vdots & J_{y_{p-1}}^p & -M_p \\ \hline \nabla_x \overline{f}^T & \nabla_{y_1} \overline{f}^T & \cdots & \cdots & \nabla_{y_p} \overline{f}^T \end{array}\right) = \begin{pmatrix} \hat{J}_x^E & \hat{J}_y^E \\ \nabla_x \overline{f}^T & \nabla_y \overline{f}^T \end{pmatrix} \tag{4.12}$$

then the gradient of f satisfies:

$$\nabla_x f^T = \nabla_x \overline{f}^T - \nabla_y \overline{f}^T \left(\hat{J}_y^E\right)^{-1} \hat{J}_x^E \tag{4.13}$$

Computing the gradient in this way requires work on the order of $\chi_b(J_E) \cdot \omega(f)$ [13] and space on the order of $\omega(f) = \omega(\overline{f}) + \sum_{i=1}^{p} \omega(F_i)$; whereas direct application of reverse mode AD has the same space requirement and only requires work proportional to $\omega(f)$ [23]. Since $\chi_b(J_E) \geq 1$, it appears as if reverse mode AD outperforms the method outlined in Figure 4.1 and (4.11) - (4.13). However, as we will see in **Section** 4.2 we can use the structure of f to match the work complexity of reverse mode, and greatly reduce the memory required. This is important, as the reverse mode requirement to save the entire computational tape can be prohibitively expensive memory-wise for large-scale problems.

We can extend the structured method to the Hessian, by differentiating the process by which the gradient is determined; similarly to that which is outlined in **Section** 2.4.3. If we represent the structured computation in Figure 4.1 as $F^E(x, y) = 0$, we can write the gradient process as [14]:

$$\begin{array}{ll} \text{Solve and differentiate to obtain } J_E: & F^E(x, y) = 0 \\ \text{Solve for } w: & (\hat{J}_y^E)^T w + \nabla_y \overline{f}^T = 0 \\ \text{Solve for } \nabla f: & (\hat{J}_x^E)^T w + \nabla_x \overline{f}^T - \nabla f^T = 0 \end{array} \tag{4.14}$$

Then differentiating (4.14) with respect to (x, y, w) yields the extended Hessian:

$$H_E = \begin{pmatrix} \hat{J}_x^E & \hat{J}_y^E & 0 \\ \left(\hat{J}_{yx}^E\right)^T w + \nabla_{yx}^2 \overline{f} & \left(\hat{J}_{yy}^E\right)^T w + \nabla_{yy}^2 \overline{f} & \left(\hat{J}_y^E\right)^T \\ \left(\hat{J}_{xx}^E\right)^T w + \nabla_{xx}^2 \overline{f} & \left(\hat{J}_{xy}^E\right)^T w + \nabla_{xy}^2 \overline{f} & \left(\hat{J}_x^E\right)^T \end{pmatrix} \tag{4.15}$$

If we define a function:

$$g(x, y) = \overline{f}(x, y) + \sum_{i=1}^{p} w_i^T F_i(x, y) \tag{4.16}$$

then we can partition the extended Hessian as follows:

$$H_E = \left(\begin{array}{c|cc} \hat{J}_x^E & \hat{J}_y^E & 0 \\ \nabla_{yx}^2 g & \nabla_{yy}^2 g & (\hat{J}_y^E)^T \\ \hline \nabla_{xx}^2 g & \nabla_{xy}^2 g & (\hat{J}_x^E)^T \end{array}\right) = \begin{pmatrix} A & L \\ B & M \end{pmatrix} \tag{4.17}$$

We can also achieve symmetry in the extended Hessian by block permutations [14]:

$$H_E^S = \begin{pmatrix} 0 & \hat{J}_y^E & \hat{J}_x^E \\ (\hat{J}_y^E)^T & \nabla_{yy}^2 g & \nabla_{yx}^2 g \\ (\hat{J}_x^E)^T & \nabla_{xy}^2 g & \nabla_{xx}^2 g \end{pmatrix} \tag{4.18}$$

The Hessian matrix H can then be recovered from the partitioned extended Hessian in (4.17) by the Schur complement computation:

$$H = B - ML^{-1}A \tag{4.19}$$

This structured method for computing the Hessian requires work proportional to $\chi_\sigma(H_E) \cdot \omega(f)$, and space proportional to $\omega(f) = \omega(\overline{f}) + \sum_{i=1}^{p} \omega(F_i)$. While using reverse-mode AD straightforwardly has the same space requirement and needs work proportional to $n \cdot \omega(f)$. The relationship between n and $\chi_\sigma(H_E)$ depends on the sparsity of H_E, which is problem-specific so we cannot conclude which is more efficient. However, we will show in **Section** 5 that if we approach (4.19) in a certain way, we can significantly reduce the memory required to compute the Hessian.

4.2 Structured Gradient Computation

For large-scale problems, gradient computation by reverse-mode AD is much more efficient than forward mode with respect to time complexity as $m = 1 \ll n$ [23]. However, reverse mode requires the entire computational tape to be saved and this can become an infeasibly large storage requirement. If the machine being used to do this computation runs out of fast memory, having to save the tape to secondary memory can have negative effects on computation time as well [9].

However, when the function to be differentiated exhibits structure as in Figure 4.1, we can compute the gradient in such a way as to vastly reduce the storage required. The key to this method is that we can recover the gradient $\nabla f(x)$, without ever having to explicitly compute the derivative matrices $\hat{J}_x^E$, $\hat{J}_y^E$ [17]. To see this we examine the computation in (4.13), which is required to recover the gradient from the extended Jacobian J_E. Computing (4.13) by reverse mode is very similar to the process outlined in **Section** 2.4.2. We need to compute $T = \nabla_y \overline{f}^T \left(\hat{J}_y^E\right)^{-1} \hat{J}_x^E$, the first step of which is to determine the product:

$$W = \nabla_y \overline{f}^T \left(\hat{J}_y^E\right)^{-1} \tag{4.20}$$

which we rewrite in the following more convenient form:

$$\left(\hat{J}_y^E\right)^T W^T = \nabla_y \overline{f} \tag{4.21}$$

This corresponds to the upper triangular system [17]:

$$\begin{pmatrix} -M_1 & \left(J_{y_1}^2\right)^T & \left(J_{y_1}^3\right)^T & \cdots & \left(J_{y_1}^{p-1}\right)^T & \left(J_{y_1}^p\right)^T \\ & -M_2 & \left(J_{y_2}^3\right)^T & \cdots & \vdots & \left(J_{y_2}^p\right)^T \\ & & -M_3 & \cdots & \vdots & \vdots \\ & & & \ddots & \left(J_{y_{p-2}}^{p-1}\right)^T & \left(J_{y_{p-2}}^p\right)^T \\ & & & & -M_{p-1} & \left(J_{y_{p-1}}^p\right)^T \\ & & & & & -M_p \end{pmatrix} \begin{pmatrix} w_1^T \\ w_2^T \\ w_3^T \\ \vdots \\ w_{p-1}^T \\ w_p^T \end{pmatrix} = \begin{pmatrix} \nabla_x \overline{f} \\ \nabla_{y_1} \overline{f} \\ \nabla_{y_2} \overline{f} \\ \vdots \\ \nabla_{y_{p-1}} \overline{f} \\ \nabla_{y_p} \overline{f} \end{pmatrix} \tag{4.22}$$

then

$$T = \nabla_y \overline{f}^T \left(\hat{J}_y^E \right)^{-1} \hat{J}_x^E = W \cdot \hat{J}_x^E = w_1 J_x^1 + w_2 J_x^2 + \ldots + w_{p-1} J_x^{p-1} + w_p J_x^p \tag{4.23}$$

Note that since we are dealing with a scalar-valued function $\left(\text{ie. } m = 1\right)$, $w_i, \nabla_{y_i} \overline{f} \in \mathbb{R}^{n_i}$, $i = 1{:}\, p$ are vectors, but otherwise would be matrices with corresponding notation W_i, $\overline{J}_{y_i}$, for $m > 1$.

Solving (4.22) requires working through the w_i's from bottom to top. The first few w_i's are calculated as follows:

$$\begin{aligned} w_p: \quad -M_p w_p^T &= \nabla_{y_p} \overline{f} \\ w_p &= -M_p^{-1} \nabla_{y_p} \overline{f}^T \end{aligned} \tag{4.24}$$

$$\begin{aligned} w_{p-1}: \quad -M_{p-1} w_{p-1}^T + \left(J_{y_{p-1}}^p \right)^T w_p^T &= \nabla_{y_{p-1}} \overline{f} \\ w_{p-1} &= M_{p-1}^{-1} \left(-\nabla_{y_{p-1}} \overline{f}^T + w_p J_{y_{p-1}}^p \right) \end{aligned} \tag{4.25}$$

$$\begin{aligned} w_{p-2}: \quad -M_{p-2} w_{p-2}^T + \left(J_{y_{p-2}}^{p-1} \right)^T w_{p-1}^T + \left(J_{y_{p-2}}^p \right)^T w_p^T &= \nabla_{y_{p-2}} \overline{f} \\ w_{p-2} &= M_{p-2}^{-1} \left(-\nabla_{y_{p-2}} \overline{f}^T + w_{p-1} J_{y_{p-2}}^{p-1} + w_p J_{y_{p-2}}^p \right) \end{aligned} \tag{4.26}$$

Note that in each w_i, we only require blocks from $\hat{J}_y^E$ in the form of matrix-matrix products, so we can avoid computing $\hat{J}_y^E$ explicitly.

Then T can be computed by explicitly differentiating the final intermediate function $\overline{f}$ and then solving for the w_i's as in (4.24) - (4.26). The computing process is outlined as follows [17]:

1. $w_p = -M_p^{-1} \nabla_{y_p} \overline{f}^T$
2. Use reverse-mode AD on $F_p\left(x, y_1, \ldots, y_{p-1}\right)$ with seed matrix w_p^T to determine

   $$w_p J_x^p \quad \text{and} \quad w_p J_{y_1}^p, \ldots, w_p J_{y_{p-1}}^p$$

3. $w_{p-1} = M_{p-1}^{-1} \left(-\nabla_{y_{p-1}} \overline{f}^T + w_p J_{y_{p-1}}^p \right)$

4. Use reverse-mode AD on $F_{p-1}\left(x, y_1, \ldots, y_{p-2}\right)$ with seed matrix w_{p-1}^T to determine

$$w_{p-1} J_x^{p-1} \quad \text{and} \quad w_{p-1} J_{y_1}^{p-1}, \ldots, w_{p-1} J_{y_{p-2}}^{p-1}$$

5. $w_{p-2} = M_{p-2}^{-1}\left(-\nabla_{y_{p-2}} \overline{f}^T + w_{p-1} J_{y_{p-2}}^{p-1} + w_p J_{y_{p-2}}^{p}\right)$

6. Continue process back until $p = 1$

Note that we also only require $\hat{J}_x^E$ in the form of matrix-matrix products, and so it does not need to be calculated explicitly either.

We now present the algorithm for structured gradient computation [17]. It can be represented as a 3-step algorithm. The first step is to calculate all of the intermediate variables $y_1, y_2, \ldots, y_p$ from top to bottom. The second step is to differentiate the final intermediate function $\overline{f}$ to get the derivative vectors $\left(\nabla_x \overline{f}^T, \nabla_{y_1} \overline{f}^T, \ldots, \nabla_{y_p} \overline{f}^T\right)$ by reverse-mode AD. The final step computes the gradient by (4.13), implementing steps 1 through 6 above in an efficient way.

Algorithm Structure-2 (S-2)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$

Outputs: Function value $z = f(x)$, and gradient $\nabla_x f \in \mathbb{R}^n$

1. Following Figure 4.1 evaluate $y_1, y_{2,...,} y_p$
2. Evaluate $z = \overline{f}(x, y_1, ..., y_p)$ and apply reverse-mode AD to $\overline{f}$ to obtain $\nabla \overline{f}^T = (\nabla_x \overline{f}^T, \nabla_{y_1} \overline{f}^T, ..., \nabla_{y_p} \overline{f}^T)$.
3. I) Initialize: $v_i = 0, i = 1{:}\, p.\ \nabla_x f = \nabla_x \overline{f}$

   II) For $j = p, p-1, ..., 1$

   Solve for w_j : $M_j w_j = \nabla_{y_j} \overline{f} - v_j$

   Evaluate $F_j(x, y_1, ..., y_{j-1})$ and apply reverse-mode AD with vector w_j to get $w_j^T \cdot (J_x^j, J_{y_1}^j, ..., J_{y_{j-1}}^j)$. Set $v_i^T = v_i^T + w_j^T \cdot J_{y_i}^j \ \ (i = 1, ..., j-1)$

   Update: $\nabla_x f^T \leftarrow \nabla_x f^T + w_j^T J_x^j$

Figure 4.2: Algorithm S-2

Analyzing the complexity of *S-2*, we have that step 1 requires time proportional to $\sum_{i=1}^{p} \omega(F_i)$ and space proportional to $\sigma(F)$. Step 2 requires a reverse mode AD computation, and since we are dealing with a scalar valued underlying function, the time and space required are proportional to $\omega(\overline{f})$. Step 3 requires p reverse mode AD computations, and since the w_i's are vectors, this requires time proportional to $\sum_{i=1}^{p} \omega(F_i)$ and space proportional to $\max\{\omega(F_i),\ i = 1{:}\, p\}$. Putting all this together, we have that the *S-2* algorithm requires time and space:

$$\omega_{S-2} = \omega\left(\overline{f}\right) + \sum_{i=1}^{p} \omega\left(F_i\right) = \omega\left(f\right) \tag{4.27}$$

$$\sigma_{S-2} = \max\left\{\omega(\overline{f}),\ \omega(F_i),\ i = 1{:}\, p\right\} \tag{4.28}$$

Compare this to applying reverse mode AD to the underlying function, where the time and space requirements are $\omega_{rm} = \omega(f)$ and $\sigma_{rm} = \omega(f)$. The *S-2* algorithm has the same time complexity as reverse mode AD, but since $\omega(f) = \omega\left(\overline{f}\right) + \sum_{i=1}^{p} \omega\left(F_i\right)$, we have that $\sigma_{S-2} \ll \sigma_{rm}$ [17]. Since we only use reverse mode AD on one intermediate function at a time, rather than the whole underlying function; the space required is greatly reduced without sacrificing efficiency in computing time.

5 Structured Hessian Computation

One of the main contributions of this thesis is an extension of the ideas involved in developing Algorithm *S-2* for gradient computation [17], and applying them to Hessian computation. We will consider three methods for computing the Hessian matrix when the function $z = f(x)$ is given in the form of Figure 4.1: an implicit method very similar to Algorithm *S-2*, an explicit method where sparsity is exploited, but the extended Jacobian matrices $\hat{J}_x^E$, $\hat{J}_y^E$ are computed, and a gradient differencing method where *S-2* is used to compute the gradients.

First, the equation for Hessian computation

$$H = B - ML^{-1}A \tag{5.1}$$

must be presented in a more tractable form. Note that we can write

$$L = \begin{pmatrix} \hat{J}_y^E & 0 \\ \nabla^2_{yy} g & (\hat{J}_y^E)^T \end{pmatrix} = \begin{pmatrix} I & 0 \\ \nabla^2_{yy} g (\hat{J}_y^E)^{-1} & (\hat{J}_y^E)^T \end{pmatrix} \cdot \begin{pmatrix} \hat{J}_y^E & 0 \\ 0 & I \end{pmatrix} \tag{5.2}$$

and both matrices in the product on the right hand side are nonsingular. Therefore

$$L^{-1} = \begin{pmatrix} \hat{J}_y^E & 0 \\ 0 & I \end{pmatrix}^{-1} \cdot \begin{pmatrix} I & 0 \\ \nabla^2_{yy} g (\hat{J}_y^E)^{-1} & (\hat{J}_y^E)^T \end{pmatrix}^{-1} \tag{5.3}$$

and

$$ML^{-1}A = (M \begin{pmatrix} \hat{J}_y^E & 0 \\ 0 & I \end{pmatrix}^{-1}) \cdot (\begin{pmatrix} I & 0 \\ \nabla^2_{yy} g (\hat{J}_y^E)^{-1} & (\hat{J}_y^E)^T \end{pmatrix}^{-1} A) \tag{5.4}$$

Then from (5.1) we have:

$$H = B - ML^{-1}A = \nabla^2_{xx} g - \left(\nabla^2_{xy} g \cdot \left(\hat{J}_y^E\right)^{-1}\right) \cdot \hat{J}_x^E - \left(\hat{J}_x^E\right)^T \cdot \left[\left(\hat{J}_y^E\right)^{-T} \left[\nabla^2_{yx} g - \left(\nabla^2_{yy} g\right)\left(\hat{J}_y^E\right)^{-1} \hat{J}_x^E\right]\right] \tag{5.5}$$

5.1 Implicit Method

Extending the structured gradient approach from **Section** 4.2, we propose a new method to solve for the Hessian *H* of *f* without explicitly computing $\left(\hat{J}_x^E, \hat{J}_y^E\right)$. Our approach is based on the following observation.

Observation 1: Let *V* be a matrix with *r* columns and row dimension equal to the size of the intermediate vector *y* from Figure 4.1. Assume $z = f(x)$ is a scalar-valued function with structure defined by (1) and let lower triangular matrix $\hat{J}^E$ be defined by (4.11). Then the matrix $W = (\hat{J}_y{}^E)^{-T} V$ can be computed w/o precomputing matrix $\hat{J}_y^E$. Specifically, the block-rows of W are computed where the computation of block W_j involves the application of reverse-mode AD to function F_j with matrix W_j. The computing time is proportional to $r \cdot \omega(f)$ and the space required is proportional to $\max_i \{\omega(F_i)\}$ [17].

Algorithm Implicit-Inverse-Product (IIP)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$, and matrix $V \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times r}$

Outputs: Matrix $W \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times r}$ satisfying $W = (\hat{J}_y{}^E)^{-T} V$

1. Following Figure 4.1, evaluate $y_1, y_{2,\ldots,} y_p$
2. I) Initialize: $T_i = 0,\ i = 1{:}\ p$

   II) For $j = p, p-1, \ldots, 1$

   Solve for W_j : $M_j W_j = T_j^T - V_j$

   Evaluate $F_j(x, y_1, \ldots, y_{j-1})$ and apply reverse-mode AD with vector W_j to

   Get $W_j^T \cdot (J_{y_1}^j, \ldots, J_{y_{j-1}}^j)$.

   Set $T_i = T_i + W_j^T \cdot J_{y_i}^j \ \ (i = 1, \ldots, j-1)$

Figure 5.1: Algorithm IIP

Algorithm *IIP* shows how Observation 1 can be implemented, since W has r columns the total time cost is:

$$\omega \sim r \cdot \sum_{i=1}^{p} \omega(F_i) \leq r \cdot \omega(f) \tag{5.6}$$

Since we only use reverse-mode AD on one intermediate function at a time, the space cost is:

$$\sigma \sim \max_i \{\omega(F_i)\} \tag{5.7}$$

Due to form (5.5), we can solve for H without explicitly computing $\hat{J}_y^E$, through repeated use of *IIP* according to the following algorithm, where we have included the time and space requirements for each step:

Algorithm Implicit-Compute-Hessian (ICH)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$

Outputs: Function value $z = f(x)$, gradient $\nabla_x f \in \mathbb{R}^n$, and Hessian $\nabla^2 f \in \mathbb{R}^{n \times n}$

1. Evaluate $y_1, y_{2,\ldots,} y_p$

   $\left[\omega \sim \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

2. Solve $(\hat{J}_y^E)^T w = -\nabla_y \overline{f}$ to obtain *w*, using *S-2*

   $\left[\omega \sim \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

3. Let $g(x,y) = \overline{f}(x,y) + \sum_{i=1}^{p} w_i^T F_i(x,y)$ and compute its Hessian

   $\nabla^2 g = \begin{bmatrix} \nabla^2_{xx} g & \nabla^2_{xy} g \\ \nabla^2_{yx} g & \nabla^2_{yy} g \end{bmatrix} = \nabla^2 \overline{f} + \sum_{i=1}^{p} \nabla^2 \left(w_i^T F_i \right)$ using sparse AD.

   $\left[\omega \sim \chi\left(\nabla^2 \overline{f}\right) \cdot \omega\left(\overline{f}\right) + \sum_{i=1}^{p} \chi\left(\nabla^2 w_i^T F_i\right) \cdot \omega(F_i),\ \sigma \sim \max_i \left\{ \omega(F_i), \omega\left(\overline{f}\right) \right\} \right]$

4. Using *IIP*, compute $R^T = \left(\hat{J}_y^E\right)^{-T} \left(\nabla^2_{xy} g\right)^T$ $\left[\omega \sim n \cdot \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

5. Using *IIP*, compute $C^T = \left(\hat{J}_y^E\right)^{-T} \left(\nabla^2_{yy} g\right)^T$

   $\left[\omega \sim \sum_{i=1}^{p} n_i \cdot \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

6. Using reverse-mode AD, compute $T = C \cdot \hat{J}_x^E = C_1 J_x^1 + \ldots + C_p J_x^p$

   $\left[\omega \sim \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

7. Using *IIP*, compute $S = \left(\hat{J}_y^E\right)^{-T} \left[\nabla^2_{yx} g - T \right]$

   $\left[\omega \sim \sum_{i=1}^{p} n_i \cdot \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

8. Using reverse-mode AD, compute $\sum_{i=1}^{p} R_i J_x^i$ and $\sum_{i=1}^{p} S_i^T J_x^i$

   $\left[\omega \sim \sum_{i=1}^{p} n_i \cdot \omega(f),\ \sigma \sim \max_i \{\omega(F_i)\} \right]$

9. Set $\nabla^2 f = \nabla^2_{xx} g - \sum_{i=1}^{p} R_i J_x^i - \sum_{i=1}^{p} S_i^T J_x^i$

Figure 5.2: Algorithm ICH

So from *ICH*, it is clear that our method requires time and space:

$$\begin{aligned}\omega &\sim \left(n+\sum_{i=1}^{p} n_i\right)\cdot\omega(f)+\sum_{i=1}^{p}\left[\chi\left(\nabla^2(w_i^T F_i)\right)\cdot\omega(F_i)\right]+\chi\left(\nabla^2\bar{f}\right)\cdot\omega(\bar{f})\\ \sigma &\sim \max_i\left\{\omega(F_i),\omega\left(\bar{f}\right)\right\}\end{aligned} \tag{5.8}$$

This time complexity comes from the fact that $C\in\mathbb{R}^{\sum_{i=1}^{n} n_i\times\sum_{i=1}^{n} n_i}$, and $\nabla^2_{yx}g$, $T\in\mathbb{R}^{n\times\sum_{i=1}^{n} n_i}$, and so by Observation 1, steps 5 and 7 have their corresponding complexity. We can compare this to the time and space required in using sparse AD without structure:

$$\omega\sim\chi_\sigma(\nabla^2 f)\cdot\omega(f) \text{ and } \sigma\sim\omega(f) \tag{5.9}$$

Due to how the general structure is defined, we know that $\max_i\left\{\omega(F_i),\omega\left(\bar{f}\right)\right\}\le\omega(f)$ so *ICH* requires less space, and depending on the specific structure, potentially considerably less space than sparse AD. However the structured method does incur a loss in efficiency with respect to time, as $\chi_\sigma(\nabla^2 f)\le n<\sum_{i=1}^{p} n_i$. So we will have to see if this trade off of time for memory efficiency is worthwhile in some numerical experiments.

5.2 Explicit Method

In analyzing the *ICH* algorithm, it is apparent that large matrix-Jacobian products must be computed by AD (eg. Step 5 of *ICH* requires the product of two $\sum_{i=1}^{p} n_i\times\sum_{i=1}^{p} n_i$ matrices) and this is responsible for Algorithm *ICH*'s relatively high time complexity. So we also consider a more explicit approach to computing the exact Hessian that avoids using AD to calculate these large products. Again using the form for the Hessian given in (5.5), now we first explicitly compute the extended Jacobian matrices $\hat{J}_x^E,\hat{J}_y^E$, then directly calculate the matrix products as needed.

This method requires modified versions of the *S-2* and *IIP* algorithms to account for the direct matrix product calculations:

Algorithm Explicit Structure-2 (ES-2).

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$, matrices $\hat{J}_x^E \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times n}$ and $\hat{J}_y^E \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times \sum_{i=1}^{p} n_i}$

Outputs: Function value $z = f(x)$, and gradient $\nabla_x f \in \mathbb{R}^n$

1. Following Figure 4.1, evaluate $y_1, y_{2,...,} y_p$
2. Evaluate $z = \overline{f}(x, y_1, ..., y_p)$ and apply reverse-mode AD to $\overline{f}$ to obtain $\nabla \overline{f}^T = (\nabla_x \overline{f}^T, \nabla_{y_1} \overline{f}^T, ..., \nabla_{y_p} \overline{f}^T)$.
3. I) Initialize: $v_i = 0,\ i = 1{:}\ p.\ \nabla_x f = \nabla_x \overline{f}$

   II) For $j = p, p-1, ..., 1$

   Solve for w_j: $M_j w_j = \nabla_{y_j} \overline{f} - v_j$

   Set $v_i^T = v_i^T + w_j^T \cdot J_{y_i}^j \ \ (i = 1, ..., j-1)$

   Update: $\nabla_x f^T \leftarrow \nabla_x f^T + w_j^T J_x^j$

Figure 5.3: Algorithm ES-2

Algorithm Explicit-Inverse-Product (EIP) For $W = (\hat{J}_y^{\ E})^{-T} V$

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$, matrices $V \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times r}$ and $\hat{J}_y^E \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times \sum_{i=1}^{p} n_i}$

Outputs: Matrix $W \in \mathbb{R}^{\sum_{i=1}^{p} n_i \times r}$ satisfying $W = (\hat{J}_y^{\ E})^{-T} V$

1. Following Figure 4.1, evaluate $y_1, y_{2,...,} y_p$
2. I) Initialize: $T_i = 0,\ i = 1{:}\ p$

   II) For $j = p, p-1, ..., 1$

   Solve for W_j: $M_j W_j = T_j^{\ T} - V_j$

   Set $T_i = T_i + W_j^T \cdot J_{y_i}^j \ \ (i = 1, ..., j-1)$

Figure 5.4: Algorithm EIP

This leads to our explicit Hessian computation algorithm:

Algorithm Explicit-Compute-Hessian (ECH)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$

Outputs: Function value $z = f(x)$, gradient $\nabla_x f \in \mathbb{R}^n$, and Hessian $\nabla^2 f \in \mathbb{R}^{n \times n}$

1. Evaluate $y_1, y_{2,...,} y_p$
2. For $i = 1, 2, ..., p$

   Compute $J^i_{y_j}$, J^i_x $(j = 1, ..., i-1)$ using direct bicoloring AD.
3. Solve $(\hat{J}^E_y)^T w = -\nabla_y \overline{f}$ to obtain *w*, using *ES-2*.
4. Let $g(x,y) = \overline{f}(x,y) + \sum_{i=1}^{p} w_i^T F_i(x,y)$ and compute its Hessian

   $$\nabla^2 g = \begin{bmatrix} \nabla^2_{xx} g & \nabla^2_{xy} g \\ \nabla^2_{yx} g & \nabla^2_{yy} g \end{bmatrix} = \nabla^2 \overline{f} + \sum_{i=1}^{p} \nabla^2 \left(w_i^T F_i \right)$$ using sparse AD.
5. Using *EIP*, compute $R^T = \left(\hat{J}^E_y \right)^{-T} \left(\nabla^2_{xy} g \right)^T$ and $C^T = \left(\hat{J}^E_y \right)^{-T} \left(\nabla^2_{yy} g \right)^T$
6. Compute $T = C \cdot \hat{J}^E_x = C_1 J^1_x + \ldots + C_p J^p_x$
7. Using *EIP*, compute $S = \left(\hat{J}^E_y \right)^{-T} \left[\nabla^2_{yx} g - T \right]$
8. $\nabla^2 f = \nabla^2_{xx} g - \sum_{i=1}^{p} R_i J^i_x - \sum_{i=1}^{p} S_i^T J^i_x$

Figure 5.5: Algorithm ECH

The most expensive components of the *ECH* algorithm are the blockwise computations of the extended Jacobian $(J^i_{y_j}, J^i_x)$, as well as computing the Hessians of the intermediate functions $(F_i, \overline{f})$. The time and space costs of the latter are given in Algorihm *ICH*, while each blockwise extended Jacobian computation requires time and space:

$$\omega \sim \chi_b(J^i_{y_j}) \cdot \omega(F_i) \text{ and } \sigma \sim \omega(F_i) + \left| J^i_{y_j} \right|_{nnz} \tag{5.10}$$

where $\left| J^i_{y_j} \right|_{nnz}$ represents the space required to store the nonzeros of $J^i_{y_j}$. The total time and space requirements for *ECH* are:

$$\begin{aligned}\omega &\sim \sum_{i=1}^{p}\left[\chi_b(J_x^i)+\sum_{j=1}^{i}\chi_b(J_{y_j}^i)+\chi_\sigma\left(\nabla^2(w_i^T F_i)\right)\right]\cdot\omega(F_i)+\chi_\sigma\left(\nabla^2\bar{f}\right)\cdot\omega(\bar{f})\\ \sigma &\sim \left|J_E\right|_{nnz}+\max_i\left\{\omega(F_i),\omega\left(\bar{f}\right)\right\}\end{aligned} \tag{5.11}$$

Comparing the costs for *ECH* to that of unstructured sparse AD in (5.9), its unclear if *ECH* has lower time complexity as execution time depends on the sparsity of all the functions involved. However, as with *ICH*, *ECH* is much more efficient memory-wise.

Comparing the costs of *ECH* to *ICH*, in terms of memory the explicit method is slightly more costly as it requires the entire extended Jacobian to be stored, unlike the implicit method. In terms of computing time, due to the exploitation of sparsity the explicit method is at worst equivalent to the implicit method, and at best significantly more efficient, depending on how sparse the intermediate functions $\left(F_i,\ i=1:p\right)$ are.

5.3 Gradient Differencing

If we do not require the exact Hessian matrix, we can use a structured forward finite differencing approach to produce a good approximation. We use forward differencing rather than central because since we are looking for an approximation to the Hessian, speed in the computation is prioritized over accuracy. Gradient evaluations are relatively inexpensive, so we choose to difference gradient evaluations in order to piece together the Hessian.

For a function $f\left(x\right):\mathbb{R}^n\to\mathbb{R}$, a gradient forward difference is presented as:

$$\nabla^2 f\left(x\right)\cdot d=\frac{\nabla f(x+hd)-\nabla f(x)}{h} \tag{5.12}$$

where *d* is a chosen direction vector. If sparsity in the Hessian is ignored, the Hessian can still always be recovered with n differences. Differencing is done column-wise, so this would require setting $d_i=e_i$, where e_i represents the i^{th} column of the identity matrix, and thus each difference would correspond to one column of the Hessian [32].

If we take sparsity into account, and have the sparsity structure of $\nabla^2 f\left(x\right)$ in hand, finding a suitable set of direction vectors can be done by finding a near-optimal coloring of the

column intersection graph associated with $\nabla^2 f(x)$. The type of coloring required is dependent on whether a direct [10, 32], or substitution [6, 21] method is desired, as described in **Section** 3.2.1. This is because differencing can only be done column-wise, and so is analogous to forward mode AD, but no such differencing analog exists for reverse mode AD; which means neither symmetric nor bicoloring methods can be used here.

What makes our approach novel is that instead of performing these gradient evaluations in a traditional way, we do so using the structured gradient approach outlined in algorithm *S-2* [17]. We outline this structured gradient forward differencing method in the following algorithm:

Algorithm Structured Forward Difference Hessian (SFDH)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$

Outputs: Function value $z = f(x)$, and gradient $\nabla_x f \in \mathbb{R}^n$, and Hessian $\nabla^2 f \in \mathbb{R}^{n \times n}$

1. Determine suitable set of direction vectors $d_1, d_2, \ldots, d_k$ (where $k \approx \chi\left(\nabla^2 f(x)\right)$)
2. Using *S-2*, compute $\nabla f(x)$
3. For $j = 1, 2, \ldots, k$
    a. Using *S-2*, compute $\nabla f(x + hd_i)$
    b. Compute forward difference $\nabla^2 f(x) \cdot d_i = \dfrac{\nabla f(x + hd_i) - \nabla f(x)}{h}$
    c. Recover the calculated components of $\nabla^2 f(x)$

Figure 5.6: Algorithm SFDH

The dominating component of *SFDH* with respect to time and memory is the structured gradient computation. From the preceding structured gradient section, we have that one such computation requires work proportional to $\omega(f)$, and space proportional to $\max\left\{\omega(\overline{f}), \omega(F_i),\ i = 1, \ldots p\right\}$.

SFDH requires $k+1$ structured gradient computations, so it has time and space costs of:

$$\omega \sim (k+1) \cdot \omega(f) \text{ and } \sigma \sim \max_i \left\{\omega(F_i), \omega(\overline{f})\right\} \tag{13}$$

We can see that this method, like *ICH* and *ECH*, has a significantly smaller space requirement than sparse AD; while also having a time requirement that is comparable to that of sparse AD. In the following experiments we compare this method to using sparse AD for the gradient evaluations to see if there are any realized advantages in computing time.

5.4 Test Functions

In the subsequent numerical experiments, we consider three test problems for the structured Hessian computation. There are two extreme cases: the dynamic system and the generalized partially separable problem. In the dynamic system each intermediate variable depends only on the intermediate variable preceding it, while in the general partially separable problem each intermediate variable depends only on the input variable *x*. The third problem represents a mix of these two extreme cases (Coleman and Xu).

5.4.1 Dynamic System

The dynamic system $z = f(x)$ is defined structurally as follows:

$$\begin{aligned} & y_0 = x \\ \text{Solve for } y_i\text{:} \quad & y_i - S(y_{i-1}) = 0 \qquad \text{for } i = 1, 2, \ldots, p \\ \text{Solve for z:} \quad & \overline{f}(x, y_1, \ldots, y_p) - \mathrm{z} = 0 \end{aligned} \tag{5.14}$$

For our experiments, the intermediate function *S* is defined as the Broyden function [5]:

$$\begin{aligned} & \textit{for } \ \mathrm{j} = 1 : \textit{inner } p \\ & \quad y_1 = (3 - 2v_1)v_1 - 2v_2 + 1 \\ & \quad \textit{for } \ \mathrm{i} = 2 : \mathrm{n} - 1 \\ & \qquad y_i = (3 - 2v_i)v_i - v_{i-1} - 2v_{i+1} + 1, \quad i = 2, 3, \ldots, n-1 \\ & \quad \textit{end} \\ & \quad y_n = (3 - 2v_n)v_n - v_{n-1} + 1 \\ & \textit{end} \end{aligned} \tag{5.15}$$

where *inner p* is a variable allowing us to control the workload of the intermediate functions. The last intermediate function $\overline{f}$ is defined as the Brown function:

$$
\begin{aligned}
y_i &= \left(v_i^2\right)^{v_{i+1}^2+1} \qquad i = 1,\ldots,n-1 \\
y_i &= y_i + \left(v_{i+1}^2\right)^{v_i^2+1} \qquad i = 1,\ldots,n-1 \\
z &= \sum_{i=1}^{n} y_i
\end{aligned}
\tag{5.16}
$$

and is only dependent on the last intermediate variable, y_p.

The structure of this function's extended Jacobian is as follows:

$$
J_E = \begin{pmatrix}
X & X & & & & \\
 & X & X & & & \\
 & & \ddots & \ddots & & \\
 & & & \ddots & \ddots & \\
 & & & & X & X \\
 & & & & & X
\end{pmatrix}
\tag{5.17}
$$

5.4.2 Generalized Partial Separability

The generalized partially separable function $z = f\left(x\right)$ is defined as follows:

$$
\begin{aligned}
&\text{Solve for } y_i\text{:} \quad F_i\left(x\right) - y_i = 0 \qquad \text{for } i = 1,2,\ldots,p \\
&\text{Solve for z:} \quad \overline{f}\left(x, y_1,\ldots,y_p\right) - \mathrm{z} = 0
\end{aligned}
\tag{5.18}
$$

For our experiments, each intermediate function F_i is defined as the Broyden function in (5.15), and $\overline{f}$ is defined as the Brown function in (5.16), but now it is dependent on all the intermediate variables $y_1, y_2,\ldots,y_p$ instead of just the final one [25].

The structure of this function's extended Jacobian is as follows:

$$
J_E = \begin{pmatrix}
X & X & & & & \\
X & & X & & & \\
\vdots & & & \ddots & & \\
\vdots & & & & \ddots & \\
X & & & & & X \\
 & X & X & X & X & X
\end{pmatrix}
\tag{5.19}
$$

5.4.3 General Case

We also look at a more general case, which is a cross between the dynamic system and generalized partial separability cases, where $z = f(x)$ is defined as follows:

$$\left.\begin{array}{l}
\text{Solve for } y_i\text{: } y_i - T_i(x) \quad i = 1,\ldots,6 \\
\text{Solve for } y_7\text{: } y_7 - T_7\left({}^{(y_1+y_2)}\!/_{200}\right) \\
\text{Solve for } y_8\text{: } y_8 - T_8\left({}^{(y_2+y_3+y_4)}\!/_{300}\right) \\
\text{Solve for } y_9\text{: } y_9 - T_9\left({}^{(y_5+y_6)}\!/_{200}\right) \\
\text{Solve for } y_{10}\text{: } y_{10} - T_{10}\left({}^{(y_7+y_8)}\!/_{200}\right) \\
\text{Solve for } y_{11}\text{: } y_{11} - T_{11}\left({}^{(y_8+y_9)}\!/_{200}\right) \\
\text{Solve for z: } \mathrm{z} - \bar{f}(x, y_1,\ldots,\mathrm{y}_p)
\end{array}\right\} \qquad (5.20)$$

Where $T_i(x)$ $(i = 1,\ldots,6)$ are defined as in (5.15) and $T_i(x)$ $(i = 7,\ldots,11)$ are defined as:

$$\begin{aligned}
y(1) &= \left(3 - 2x(1)\right).*x(1) - 2x(2) + 1 \\
y(i) &= \left(3 - 2x(i)\right).*x(i) - x(i-1) - 2x(i+1) + ones(n-2,1), \quad i = 2,\ldots,n-1 \\
y(n) &= \left(3 - 2x(n)\right).*x(n) - 2x(n-1) + 1
\end{aligned} \qquad (5.21)$$

And $\bar{f}$ is defined as in (5.16).

The structure of this function's extended Jacobian is as follows:

$$J_E = \begin{pmatrix} X & X & & & & & & & & & & \\ X & & X & & & & & & & & & \\ X & & & X & & & & & & & & \\ X & & & & X & & & & & & & \\ X & & & & & X & & & & & & \\ X & & & & & & X & & & & & \\ & X & X & & & & & X & & & & \\ & & X & X & X & & & & X & & & \\ & & & & & X & X & & & X & & \\ & & & & & & & X & X & & X & \\ & & & & & & & & X & X & & X \\ & & & & & X & & & & & X & X \end{pmatrix} \tag{5.22}$$

5.5 Numerical Results

We carried out experiments comparing the time and space requirements of the ICH, ECH, and SFDH algorithms to that of sparse AD for computing the Hessian matrix of each of the three testing functions. All the experiments were carried out using the ADMAT 2.0 package [35] on a computer running Windows Server 2008 with a 2.9 GHz quad-core Intel X5570 Xeon CPU, and 90 GB of RAM using Matlab R2012b.

Space requirement was measured by the largest computational tape required during each method's computation, and the space needed to store the extended Jacobian, when applicable. Since we are dealing with large problems, in most cases the space required to store the actual matrices and intermediate data is negligible relative to the size of the tape.

5.5.1 Exact Hessian

In the experiments for exact Hessian computation we used an *inner p* of 5 for the intermediate functions because for larger problem sizes, the memory required was too large to be able to increase the workload on the intermediate functions and still get results for unstructured sparse AD. In the forthcoming tables, "Sparse AD" refers to the unstructured sparse AD method

(*evalh* in ADMAT 2.0), "Implicit Structured" refers to the structured method outlined by *ICH*, and "Explicit Structured" refers to *ECH*.

Table 1: Running time (s) and memory usage (MB) for Hessian computation using sparse AD and both structured methods for the dynamic system with p = 10, inner p = 5.

n	Sparse AD	Implicit Structured	Explicit Structured
100	37.24 **10850**	15.48 **3**	4.20 **56**
200	69.03 **23076**	24.66 **15**	9.59 **153**
400	224.58 **69608**	70.19 **58**	15.61 **353**
600	308.98 **89822**	121.03 **130**	22.35 **684**
800	out of memory	216.85 **277**	21.22 **567**
1000	out of memory	319.06 **356**	35.07 **1072**
2000	out of memory	1115.86 **1243**	60.55 **1884**

Table 2: Running time (s) and memory usage (MB) for Hessian computation using sparse AD and both structured methods for the generalized partial separability problem with p = 20, inner p = 5.

n	Sparse AD	Implicit Structured	Explicit Structured
100	20.36 **7435**	16.95 **15**	10.11 **108**
200	47.93 **22295**	20.81 **43**	12.56 **119**
400	78.18 **39996**	54.29 **114**	33.70 **352**
600	128.23 **63357**	105.50 **256**	58.77 **733**
800	out of memory	187.79 **455**	52.48 **607**
1000	out of memory	205.35 **711**	65.85 **879**
2000	out of memory	491.72 **2474**	136.09 **2018**

Table 3: Running time (s) and memory usage (MB) for Hessian computation using sparse AD and both structured methods for the general problem with inner p = 5.

n	Sparse AD	Implicit Structured	Explicit Structured
100	9.55 **3659**	7.51 **51**	4.99 **51**
200	22.19 **9284**	15.05 **216**	9.24 **216**
400	42.09 **22083**	33.35 **404**	12.85 **405**
600	92.88 **48511**	63.16 **606**	16.45 **607**
800	81.43 **44161**	105.35 **707**	17.39 **708**
1000	87.65 **46402**	146.91 **766**	18.56 **768**
2000	out of memory	582.32 **2840**	46.49 **2433**

Both structured methods required significantly less memory than sparse AD, and the explicit structured method also performed much faster.

Comparing the two structured methods, Algorithm ECH performed considerably faster than *ICH*, across all three test problems. We can infer that this is because *ECH* exploits the sparsity of $\hat{J}_x^E$, $\hat{J}_y^E$ in its derivative calculations, which cannot be done for the matrix-Jacobian products in *ICH*.

When comparing memory used, we see that the difference between the two methods appears to be negligible. However in Table 1 and Table 2, we see that the space required for *ECH* scales better than *ICH* does, and so the difference in space required should increase as problem size is increased.

Comparing *ICH* to unstructured sparse AD, we see across the 3 test functions that as the problem size increases, the relative performance of *ICH* with respect to time worsens. In Table 1 and Table 2, while the structured method is faster, the gap between the two methods decreases as *n* increases. The worsening relative performance of *ICH* is most noticeable in Table 3, for example at $n = 200$ the structured method performs $\sim 30\%$ faster than sparse AD, but then at

$n = 1000$ it performs $\sim 70\%$ slower. So we can see that in all 3 cases the time required for the implicit structured method does not scale relatively well with respect to problem size.

Comparing *ECH* to unstructured sparse AD, we find that *ECH* is much more efficient with respect to time for each test problem. And unlike for *ICH*, *ECH* does not have an issue with scaling with respect to problem size. For example, in Table 3 at $n = 200$ *ECH* performs $\sim 60\%$ faster than sparse AD, then at $n = 1000$ it performs $\sim 80\%$ faster. In fact, *ECH* appears to scale better than unstructured sparse AD.

If we compare the memory required for the structured methods with that of unstructured sparse AD, we see that the structured methods vastly outperforms sparse AD. In Table 3 we have a $98 - 99\%$ reduction in memory used; while in Table 1 and Table 2 we have a reduction of $> 99\%$. This stark improvement means we can solve much larger problems with either structured approach than we could with sparse AD. As in Table 1 and Table 2, sparse AD cannot be used at $n \geq 600$ because the memory requirement is too high; whereas we could continue using the structured methods with no storage difficulties for $n >> 600$.

The results show that the structured methods yield lower memory usage, but that the implicit method can yield higher computing times at large problem sizes. However, if a restriction is placed on the amount of RAM available, having to save data to slower accessible memory could lead to gains in computation time as well. In order to examine how this would actually affect the relative computation time of the two methods, looking at the general problem we simulated restricting the RAM available to 8 GB by having any computational tape larger than 8 GB in size saved to the HDD then loaded back up again when needed.

Table 4: Running time (s) for Hessian computation using sparse AD and both structured methods for the general problem with inner p = 5, and RAM restricted to 8 GB.

n	Sparse AD	Implicit Structured	Explicit Structured
100	9.55	7.51	4.99
200	74.45	15.05	9.24
400	176.72	33.35	12.85
600	481.96	63.16	16.45
800	387.68	105.35	17.39
1000	419.01	146.91	18.56
1500	846.32	314.92	25.81

While the explicit structured method still has by far the lowest computing time, the results in Table 4 show that with a memory restriction in place, the implicit structured method performs significantly faster than sparse AD. The large increase in time for sparse AD at $n = 200$ is due to the 8 GB threshold being passed at this point. Unlike in the unrestricted case, the implicit structured method's computing time does not appear to scale any worse than sparse AD does. So we have that when a restriction is placed on memory available, both structured methods can now perform much faster than sparse AD.

5.5.2 Approximate Hessian

For approximate Hessian computation (ie. forward finite differencing) we looked only at the general case problem, this time with an *inner p* of 100, as finite differencing requires much less memory. We compared the time and space requirements for our *SFDH* algorithm, with that of a similar method where unstructured gradient computations are used instead.

Table 5: Running time (s) and memory usage (MB) for Hessian computation using unstructured and structured finite differencing method for the general problem with inner p = 100.

n	Unstructured Gradient FD	Structured Gradient FD
100	494.69 **82**	412.16 **7**
200	576.71 **136**	492.41 **10**
500	922.78 **295**	754.68 **20**
1000	1498.60 **561**	1192.60 **37**
2000	2720.20 **1092**	2088.40 **70**

The results in Table 5 show that the structured finite differencing method requires substantially less memory than unstructured finite differencing, using less than 10% of the memory the unstructured method did. The structured method also requires less computing time, $\sim 20\%$ less, and scales at a better rate than the unstructured method. If memory was restricted, the difference in computing times would be even larger. So when only an approximate Hessian is

required, the *SFDH* algorithm is more efficient with respect to time and space requirements than the corresponding unstructured method.

6 Structure Revealing Methods

In our discussions of structure thus far, we have assumed that the function to be differentiated is provided in a form analogous to that in Figure 4.1. However, this may not always be the case, as the function could be presented in such a way that its structure is not apparent. Thus, if we can automatically determine a structured form of a function, regardless of what form the function is provided in, we can utilize the structured methods described in Chapter 5 to determine Hessian matrices efficiently for a much wider range of functions.

Coleman, Xiong, and Xu [16] developed methods for determining a structured form of a function for Jacobian computation, involving inserting directed edge separators into the computational graph generated by AD in evaluating the function. We will review their work, and propose how it can be extended to use in Hessian computations.

6.1 Background

We first need to define the computational graph of a function. Consider a vector-valued function $F:\mathbb{R}^n \to \mathbb{R}^m$. In **Section** 2.1, we discussed how automatic differentiation breaks down the evaluation of a function into a partially ordered set of elementary operations, which we called the evaluation procedure of that function. The (directed) computational graph for a function evaluation, $G(F)=(V,E)$, is simply a graph representation of this evaluation procedure.

V consists of three sets of vertices such that $V=\{V_x,V_y,V_z\}$, where V_x represents the input variables $(x_1,\ldots,x_n)$, each vertex in V_y represents an elementary operation (which takes one or two inputs) and its single outputted intermediate variable, and finally V_z represents the output variables $(F_1,\ldots,F_m)$. The edge-set E represents the relative dependency of the variables. So, there exists a directed edge $e_{ij}=\left(v_{y_i},v_{y_j}\right)\in E$ if and only if the intermediate variable y_i is required by the elementary operation contained in node v_{y_j} to produce the intermediate variable y_j. We also refer to v_{y_i} as the tail node, and v_{y_j} as the head node of e_{ij}. Since the evaluation

procedure makes the function evaluation into a straight line computation, as long as F is well-defined, $G(F)$ is an acyclic graph. [16]

We then need to define a directed edge separator. $E_d \subset E$ is a directed edge separator for the graph $G(F)$, if the graph $G - \{E_d\}$ consists of two disjoint subgraphs G_1 and G_2 where all edges in E_d have the same orientation relative to G_1, G_2 [16], in other words:

$\forall e_{ij} = \left(v_{y_i}, v_{y_j}\right) \in E_d$, $v_{y_i} \in G_1$, $v_{y_j} \in G_2$.

For example, we refer back to the example function, $F : \mathbb{R}^3 \to \mathbb{R}^2$ in **Section** 2.2.1:

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin(x_3) \\ x_3 - \exp(x_1 x_2) \end{bmatrix} \tag{6.1}$$

The computational graph for this function, and the graph with a possible directed edge separator are as follows:

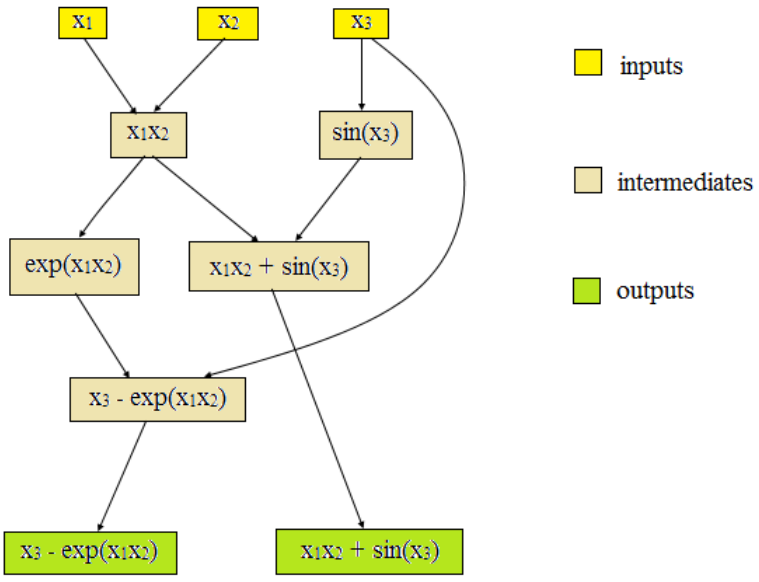

Figure 6.1: Example Computational Graph

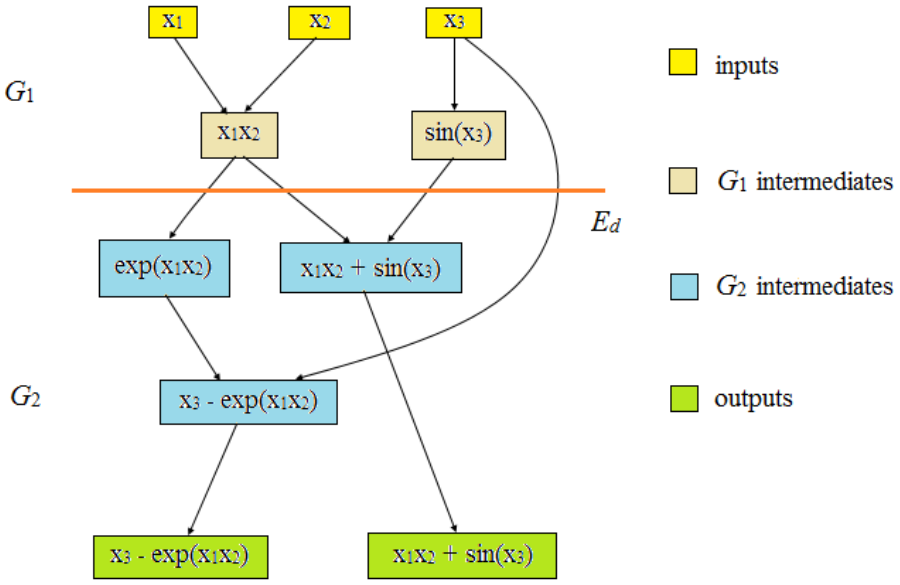

Figure 6.2: Example Computational Graph with Directed Edge Separator E_d

Then with this edge separator E_d, the function can be separated into two parts as defined by the two disjoint subgraphs G_1, G_2 of the computational graph G:

$$\begin{aligned} &\text{Solve for } y: \quad My - F_1(x) = 0 \\ &\text{Solve for output } z: \quad z - F_2(x, y) = 0 \end{aligned} \tag{6.2}$$

where y represents the intermediate variables, as defined by the tail vertices of the edges belonging to the edge separator; and $z = F(x)$.

We can generalize the application of a directed edge separator to the case where multiple (disjoint) directed separators are applied to the graph $G(F)$. We then have the directed edge separators $\{E_{d_1}, E_{d_2}, \ldots, E_{d_k}\}$, such that $G - \{E_{d_1}, E_{d_2}, \ldots, E_{d_k}\} = \{G_1, G_2, \ldots, G_{k+1}\}$; where $\{G_1, G_2, \ldots, G_{k+1}\}$ are disjoint and oriented such that $\forall e_{ij} = (v_{y_i}, v_{y_j}) \in E_{d_l}$, $v_{y_i} \in G_m$, $v_{y_j} \in G_l$ where $m < l$. Then we can break down the evaluation of $F(x)$ as follows:

$$
\begin{aligned}
&\text{Solve for } y_1: \quad M_1 y_1 - F_1(x) = 0 \\
&\text{Solve for } y_2: \quad M_2 y_2 - F_2(x, y_1) = 0 \\
&\quad \vdots \\
&\text{Solve for } y_k: \quad M_k y_k - F_k(x, y_1, y_2, \ldots, y_{k-1}) = 0 \\
&\text{Solve for output } z: \quad z - F_{k+1}(x, y_1, y_2, \ldots, y_k) = 0
\end{aligned}
\tag{6.3}
$$

where y_i is the intermediate variable defined by the tail vertices of the edge separator E_{d_i}, and is composed of nodes belonging to G_i. In (6.3), the function $F(x)$ is now in the form presented in Figure 4.1. Thus the structured techniques in Chapter 4 can be used to determine its Jacobian. If the function in question is scalar valued $\left(f : \mathbb{R}^n \to \mathbb{R}\right)$, we can then apply the structured Hessian techniques developed in Chapter 5.

6.2 Determining Separators

One way in which to determine the directed edge separators, is by what is called natural order [16]. When reverse-mode AD is used in differentiating a function, it generates a computational tape that essentially lists the elementary operations required in order of their evaluation. So, if we cut the tape at a point, this is equivalent to introducing a directed edge separator at this point.

Suppose we have the computational graph G, and the corresponding computational tape T with length $|V(G)|$, since every node in G corresponds to one elementary operation and each elementary operation corresponds to one cell in the tape. If we choose to cut the tape at the i^{th} cell, we can partition the graph G into 2 disjoint subgraphs (G_1, G_2) by defining $G_1 = G(T(1:i))$ and $G_2 = G\left(T\left(i+1:|V(G)|\right)\right)$. Since the cells in the tape are set in the order they are evaluated, all the elementary operations in G_1 will be evaluated before those in G_2. Therefore, all edges between G_1 and G_2 have their tail node in G_1, and their head node in G_2; which means that this set of edges defines a directed edge separator. [16]

Using these natural order edge separators, we can insert as many separators as we would like, allowing us to choose how long the tape is in each segment. This can be very useful as it allows the memory usage to be controlled; however selecting tape segment lengths that are too small in order to minimize storage requirements could have an adverse effect on computation time in structured computation of derivatives.

6.3 Hessian Computation

We can now apply the structure revealing technique of natural order directed edge separators to our methods for structured Hessian computation. Since the explicit method presented in **Section** 5.2, Algorithm *ECH*, was most successful we will modify it to incorporate the structure revealing capability.

We first present the modified algorithm for Hessian computation when the function $f(x)$, with corresponding computational graph $G = (V, E)$ and computational tape *T*, is not provided in structured form:

Algorithm Explicit-Structure-Revealing-Compute-Hessian (ESRCH)

Inputs: System as in Figure 4.1, vector $x \in \mathbb{R}^n$

Outputs: Function value $z = f(x)$, gradient $\nabla_x f \in \mathbb{R}^n$, and Hessian $\nabla^2 f \in \mathbb{R}^{n \times n}$

1. Choose desired tape segment length *L*.
2. Evaluate $f(x)$, and generate the tape *T*.
3. Let $p = \left\lfloor \frac{|V(G)|}{L} \right\rfloor$
4. Generate edge separators $E_{d_1}, E_{d_2}, \ldots, E_{d_p}$:

   $$E_{d_i} = E\left(T\left((i-1)\cdot L+1 : i\cdot L\right), T\left(i\cdot L+1 : (i+1)\cdot L\right)\right) \text{ for } i = 1, \ldots, p$$

5. For $i = 1, 2, \ldots, p+1$

   Generate subgraph of *G*: $G_i = G\left(T\left((i-1)\cdot L+1 : i\cdot L\right)\right)$

   Define F_i as computation corresponding to graph G_i

   Define $y_i = \left\{ v \in V_i \mid \exists e = (v, w) \in E_{d_i}, w \notin V_i \right\}$

   Compute J_x^i, $J_{y_j}^i$ $(j = 1, \ldots, i-1)$ using sparse AD.

6. Solve $(\hat{J}_y^E)^T w = -\nabla_y \overline{f}$ to obtain *w*, using *ES-2*.
7. Let $g(x, y) = \overline{f}(x, y) + \sum_{i=1}^{p} w_i^T F_i(x, y)$ and compute its Hessian

   $$\nabla^2 g = \begin{bmatrix} \nabla_{xx}^2 g & \nabla_{yx}^2 g \\ \nabla_{xy}^2 g & \nabla_{yy}^2 g \end{bmatrix} = \nabla^2 \overline{f} + \sum_{i=1}^{p} \nabla^2 \left(w_i^T F_i \right) \text{ using sparse AD.}$$

8. Using *EIP*, compute $R^T = \left(\hat{J}_y^E\right)^{-T} \left(\nabla_{xy}^2 g\right)^T$ and $C^T = \left(\hat{J}_y^E\right)^{-T} \left(\nabla_{yy}^2 g\right)^T$
9. Compute $T = C \cdot \hat{J}_x^E = C_1 J_x^1 + \ldots + C_p J_x^p$
10. Using *EIP*, compute $S = \left(\hat{J}_y^E\right)^{-T} \left[\nabla_{yx}^2 g - T\right]$
11. $\nabla^2 f = \nabla_{xx}^2 g - \sum_{i=1}^{p} R_i J_x^i - \sum_{i=1}^{p} S_i^T J_x^i$

Figure 6.3: Algorithm ESRCH

Assuming the cost of determining the edge separators is negligible relative to the rest of the computation, which is reasonable due to the technique for choosing them being relatively simple; the time and space costs for Algorithm *ESRCH* are the same as those of Algorithm *ECH*:

$$\begin{aligned}\omega &\sim \sum_{i=1}^{p}\left[\chi_b(J_x^i)+\sum_{j=1}^{i}\chi_b(J_{y_j}^i)+\chi_\sigma\left(\nabla^2(w_i^T F_i)\right)\right]\cdot\omega(F_i)+\chi_\sigma\left(\nabla^2 F_{p+1}\right)\cdot\omega(F_{p+1})\\ \sigma &\sim \left|J_E\right|_{nnz}+\max_i\left\{\omega(F_i)\right\}\end{aligned} \tag{6.4}$$

Since F_i is determined by how long we choose the tape segment to be, $\omega(F_i)\sim L$ and we can simplify (6.4) as follows:

$$\begin{aligned}\omega &\sim \sum_{i=1}^{p}\left[\chi_b(J_x^i)+\sum_{j=1}^{i}\chi_b(J_{y_j}^i)+\chi_\sigma\left(\nabla^2(w_i^T F_i)\right)\right]\cdot L+\chi_\sigma\left(\nabla^2 F_{p+1}\right)\cdot L\\ \sigma &\sim \left|J_E\right|_{nnz}+L\end{aligned} \tag{6.5}$$

The complexity results in (6.5) make it apparent how our choice of tape segment length *L*, has a large effect on the space required for the Hessian computation. The effect on computing time is not as apparent, because as *L* decreases, *p* increases as well; implying that there is an optimal length *L** to minimize the computing time. This is a subject for future research.

7 Conclusions

Automatic differentiation is a growing discipline in the field of scientific computing, as it provides methods to determine derivatives in a quick and accurate fashion. In many optimization problems, the Hessian matrix of an objective function is required, and can be determined using automatic differentiation. Much effort has been expended in determining how to exploit sparsity in a derivative matrix, in order to compute it most efficiently by automatic differentiation [2, 6, 13, 33]. However, if the Hessian is dense and does not exhibit sparsity, automatic differentiation can potentially require so much memory that completing the computation becomes infeasible [9].

The underlying structure in a function has been considered, and shown to be successful in reducing the memory requirement for gradient computation without compromising computation time efficiency [17]. Thus, in this thesis we sought to determine whether the underlying structure of a function can also be exploited to more efficiently compute its Hessian matrix by automatic differentiation, in the same manner.

Through our analysis, we have found that when a function is provided in a structured form as in Figure 4.1, this structure can be exploited to significantly reduce the space requirement in Hessian matrix computation, using either Algorithm *ECH* or *ICH*. In the case of Algorithm *ECH*, savings in computing time can be achieved as well. When a (finite-difference) approximation to the Hessian matrix is acceptable, the structured gradient differencing method outlined in Algorithm *SFDH* consistently outperformed the corresponding gradient differencing method that ignored structure with respect to computing time.

When the function to be differentiated is not provided in a structured form or the structure is not apparent, we have shown that introducing directed edge separators into the computational tape to induce a structured form of the function can yield very large savings in the space required. However, our scheme for choosing the edge separators was relatively simple: they were determined strictly based on how long each tape segment was desired to be.

This implies that future work should be done to determine methods to more optimally determine where and how frequently edge separators should be inserted into the computational

tape. Ideally, if we have a function whose structured form is known, we would like our method to choose edge separators such that this same structure is recovered. Future work should also be done on efficiently implementing the structure revealing Hessian computation method (Algorithm *ESRCH*), as well as other potentially more efficient edge separator selection techniques for Hessian computation.

References

[1] M.O. Albertson, G.G. Chappell, H.A. Kierstead, A. Kündgen, R. Ramamurthi, Coloring with no 2-colored P_4's, Electron. J. Comb. 11 (2004).

[2] B.M. Averick, J.J. Moré, C.H. Bischof, A. Carle, Computing large sparse Jacobian matrices using automatic differentiation, SIAM J. Sci. Comput. 15 (1994) 285-294.

[3] C.H. Bischof, A. Bouaricha, P. Khademi, J.J. Moré, Computing gradients in large-scale optimization using automatic differentiation, INFORMS J. Computing 9 (1997) 185-194.

[4] C.H. Bischof, P. Khademi, A. Bouaricha, A. Carle, Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation, Optim. Method Softw. 7 (1996) 1-39.

[5] C.G. Broyden, The convergence of an algorithm for solving sparse nonlinear systems, Math. Comput. 25 (1971) 285-294.

[6] T.F. Coleman, J. Cai, The cyclic problem and estimation of sparse Hessian matrices, SIAM J. Alg. Disc. Meth. 7 (1986) 221-235.

[7] T.F. Coleman, B.S. Garbow, J.J. Moré, Software for estimating sparse Hessian matrices 11 (1985) 363-378.

[8] T.F. Coleman, B.S. Garbow, J.J. Moré, Software for estimating sparse Jacobian matrices, ACM Trans. Math. Software 10 (1984) 329-345.

[9] T.F. Coleman, G.F. Jonsson, The efficient computation of structured gradients using automatic differentiation, SIAM J. Sci. Comput. 20 (1999) 1430-1437.

[10] T.F. Coleman, J.J. Moré, Estimation of sparse Hessian matrices and graph coloring problems, Math. Prog. 28 (1984) 243-270.

[11] T.F. Coleman, J.J. Moré, Estimation of sparse Jacobian matrices and graph coloring problems, SIAM J. Numer. Anal. 20 (1983) 187-209.

[12] T.F. Coleman, A. Verma, ADMIT-1: Automatic differentiation and MATLAB interface toolbox, ACM Trans. Math. Softw. 26 (2000) 150-175.

[13] T.F. Coleman, A. Verma, The efficient computation of sparse Jacobian matrices using automatic differentiation, SIAM J. Sci. Comput. 19 (1998) 1210-1233.

[14] T.F. Coleman, A. Verma, Structure and efficient Hessian calculation, in: Y. Yuan (Ed.), Advances in Nonlinear Programming, Proceedings of the 1996 International Conference on Nonlinear Programming, Kluwer Academic Publishers, Boston, MA, 1998, pp. 57-72.

[15] T.F. Coleman, A. Verma, Structure and efficient Jacobian calculation, in: M. Berz, C. Bischof, G. Corliss,A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, PA, 1996, pp. 149-159.

[16] T.F. Coleman, X. Xiong, W. Xu, Using directed edge separators to increase efficiency in the determination of Jacobian matrices via automatic differentiation, in: S. Forth, P. Hovland, E. Phipps, J. Utke,A. Walther (Eds.), Recent Advances in Algorithmic Differentiation, Springer, New York, NY, 2012, pp. 209-219.

[17] T.F. Coleman, W. Xu, The efficient evaluation of structured gradients (and undetermined Jacobian matrices) by automatic differentiation (2013).

[18] T.F. Coleman, W. Xu, Fast (structured) Newton computations, SIAM J. Sci. Comput. 31 (2008) 1175-1191.

[19] A.R. Curtis, M.J.D. Powell, J.K. Reid, On the estimation of sparse Jacobian matrices, J. Inst. Math. Appl. 13 (1974) 117-119.

[20] A.H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Review 47 (2005) 629-705.

[21] A.H. Gebremedhin, A. Tarafdar, F. Manne, A. Pothen, New acyclic and star coloring algorithms with application to computing Hessians, SIAM J. Sci. Comput. 29 (2007) 1042-1072.

[22] C.D. Good, R.I. Scahill, N.C. Fox, J. Ashburner, K.J. Friston, D. Chan, W.R. Crum, M.N. Rossor, R.S.J. Frackowiak, Automatic differentiation of anatomical patterns in the human brain: validation with studies of degenerative dementias, NeuroImage 17 (2002) 29-46.

[23] A. Griewank, Some bounds on the complexity of gradients, Jacobians, and Hessians, in: P.M. Pardalos (Ed.), Complexity in Numerical Optimization, World Scientific Publishing Co., Singapore, 1993, pp. 128-162.

[24] A. Griewank, Direct calculation of Newton steps without accumulating Jacobians, in: T.F. Coleman,Y. Li (Eds.), Large-Scale Numerical Optimization, SIAM, Philadelphia, PA, 1990, pp. 115-137.

[25] A. Griewank, P.L. Toint, On the unconstrained optimization of partially separable functions, in: M.J.D. Powell (Ed.), Nonlinear Optimization, Academic Press, London, 1981, pp. 301-312.

[26] A. Griewank, A. Walther, Evaluating Derivatives: Principles, and Techniques of Algorithmic Differentiation, 2nd ed., SIAM, Philadelphia, PA, 2005.

[27] M. Henrard, Calibration in finance: very fast greeks through algorithmic differentiation and implicit function, Procedia Comput. Sci. 18 (2013) 1145-1154.

[28] A. John Arul, N. Kannan Iyer, K. Velusamy, Efficient reliability estimate of passive thermal hydraulic safety system with automatic differentiation, Nucl. Eng. Des. 240 (2010) 2768-2778.

[29] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum, New York, NY, 1972, pp. 85-103.

[30] M.J. Krause, V. Heuveline, Parallel fluid flow control and optimisation with lattice Boltzmann methods and automatic differentiation, Comput. Fluids 80 (2012) 28-36.

[31] R. Leidenberger, K. Urban, Automatic differentiation for the optimization of a ship propulsion and steering system: a proof of concept, J. Global Optim. 49 (2010) 497-504.

[32] M.J.D. Powell, P.L. Toint, On the estimation of sparse Hessian matrices, SIAM J. Numer. Anal. 16 (1979) 1060-1074.

[33] A.K.M. Shahadat Hossain, T. Steihaug, Computing a sparse Jacobian matrix by rows and columns, Optim. Method Softw. 10 (1998) 33-48.

[34] A. Walther, Computing sparse Hessians with automatic differentiation, ACM Trans. Math. Softw. 34 (2008).

[35] Cayuga Research Inc. ADMAT-2.0 Users Guide. http://www.cayugaresearch.com/, 2009.