Efficient Jacobian Determination by Structure-Revealing Automatic Differentiation

by

Xin Xiong

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

Abstract

This thesis is concerned with the efficient computation of Jacobian matrices of nonlinear vector maps using automatic differentiation (AD). Specificially, we propose the use of two directed edge separator methods, the weighted minimum separator and natural order separator methods, to exploit the structure of the computional graph of the nonlinear system. This allows for the efficient determination of the Jacobian matrix using AD software. We will illustrate the promise of this approach with computational experiments.

Acknowledgements

I am heartily thankful to my supervisor, Thomas F. Coleman, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In the summer of 2012, the 6^{th} International Conference on Automatic Differentiation was held in Colorado State University. In spite of the fact that automatic differentiation is a joint field of computer science and mathematics, the conference attracted many scholars and researchers from a wide diversity of areas, including magnetohydrodynamics[38], atmospheric remote sensing[34], community land modelling[35], crop modelling[33], and pressure swing adsorption optimization[21].

Computational modeling and quantitative analysis is a trend for almost all disciplines nowadays. Scientific computation plays a more and more important role in industry and academia, as it helps making estimations and predictions; in some areas fast computation is essential — computing speed is the bottleneck and directly determines the overall efficiency.

In many cases the determination of Jacobian matrix reflects the major portion of the overall computation, this is where automatic differentiation comes in — given the computer code to evaluate a function, automatic differentiation can calculate the function's Jacobian matrix accurately and efficiently. Compared to the well-known finite difference method, automatic differentiation can be much more efficient, especially when the function has some special structures.

This efficiency and accuracy pay-off is why researchers and practitioners with different background joined the conference. Many of them experienced computation speed issues. Having tried different techniques to optimize the method calculating Jacobian matrices, they turned to automatic differentiation and some obtained excellent results. In the process of exploiting AD, they successfully made improvements on AD itself, and came to the conference to share their ideas.

Automatic differentiation performs extraordinarily well under many scenarios; however it has limitations. For example, reverse mode of AD cannot be used for complicated computation due to space constraint, and the superior sparsity techniques cannot be used when the Jacobian matrix is dense. Therefore it is beneficial to develop methods to overcome these limitatons, to allow automatic differentiation to be used more widely.

1.2 Structure of the Thesis

In this thesis we will discuss a graph theory approach to improve automatic differentiation. More specifically, directed edge separators of the associated computational graph are located, to help reduce the space requirement of the AD reverse mode, and reveal hidden structure for apparently dense functions.

Chapter 2 begins with a brief review of automatic differentiation, followed by the definition of computational graph and an introduction to the directed edge separator idea. **Chapter** 3 presents two methods locating good directed edge separators and the test result of the methods. Hidden structure revealing methods and sparsity techniques are discussed in **Chapter** 4, followed by numerical experiments. In **Chapter** 5 we investigate a special case that the structure of functions is provided, in which a special technique can be used to greatly boost the efficiency. Main conclusions of the thesis and some directions of future work are presented in **Chapter** 6.

Chapter 2

Automatic Differentiation

2.1 Introduction

In many scientific and engineering applications, the computations of matrices of derivatives are performed repeatedly. For example if $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a smooth differentiable mapping, then many nonlinear regression methods require the determination of the $m \times n$ Jacobian matrix $J(x) \triangleq \left(\frac{\partial f_i}{\partial x_j}\right)_{j=1:n}^{i=1:m}$, evaluated at many iterates of $x \in \mathbb{R}^n$, where $F(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}$. In a similar fashion the minimization of a sufficiently smooth nonlinear function, $f : \mathbb{R}^n \mapsto \mathbb{R}$, may require both the determination of the gradient, $\nabla f(x) = \left(\frac{\partial f}{\partial x_i}\right)_{i=1:n}$ and the $n \times n$ (symmetric) Hessian matrix, $H(x) \triangleq \left(\frac{\partial^2 f}{\partial x_i \partial x_j}\right)_{i=1:n,j=1:n}$

The repeated calculation of the derivative matrices and Hessian matrices often represent a large portion of the overall computational cost. Therefore it is valuable to develop a general method that can obtain these matrices accurately and efficiently.

Automatic differentiation (AD) is a joint field of computer science and applied mathematics, that has advanced rapidly over the past 15 years [37]. AD can compute matrices of derivatives numerically given the source code to evaluate a function F (or in the case of minimization, the objective function f). Good methods that can exploit and take advantage of sparsity, constant values, and duplicated values have also been developed [6, 40]. Furthermore, if certain structures are embedded in the objective functions, and are provided, then the efficiency of automatic differentiation can be greatly improved [3, 11, 13, 14, 15, 19, 37].

This thesis is concerned with the improvement of AD under two cases: when the structure is not provided and when the structure is provided. For example, consider the automatic differentiation of the gradient function $\nabla f(x) = \left(\frac{\partial f}{\partial x_i}\right)_{i=1:n} \in \mathbb{R}^n$, provided the source code to evaluate $f(x)$. The reverse mode[37] of automatic differentiation can be applied to get the gradient accurately and efficiently, where the computational cost of getting the derivative is same as that to evaluate the function[15]. However the space requirement of reverse mode is the same as the number of arithmetic operations needed to evaluate f, which implies that the real running time can be considerably longer than the theoretical prediction because of the need to use secondary memory[11]. One solution to this space issue is to take advantage of the structure of f [19], and apply AD 'slice by slice'. This approach is very effective but does require the user to understand the structure and provide them. In this thesis we will do further tests on this method, and discuss a more automatic but less intuitive solution.

2.2 Automatic Differentiation and The Edge Separator

Let us consider a nonlinear mapping

$$F : \mathbb{R}^n \mapsto \mathbb{R}^m$$

where $F(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}$, and each component function $f_i : \mathbb{R}^n \mapsto \mathbb{R}^1$ is differentiable. The Jacobian matrix $J(x)$ is the $m \times n$ matrix of first derivatives defined by: $J_{ij} = \frac{\partial f_i}{\partial x_j}$ $(i = 1, \cdots, m; j = 1, \cdots, n)$. If the source code to evaluate $F(x)$ is provided, automatic differentiation can be used to evaluate the Jacobian matrix $J(x)$. Generally speaking, in the presence of sparsity in $J(x)$, the work required to evaluate $J(x)$, via a *combination* of the forward and reverse modes of AD, is proportional to $\chi_B(G^D(J)) \cdot \omega(F)$, where χ_B is the bi-chromatic number of the double intersection graph $G^D(J)$, and $\omega(\cdot)$ is the work required (i.e., number of flops) to evaluate the argument -see [15]. Note that if reverse mode AD is invoked, the space required to compute the Jacobian is proportional to $\omega(F)$, and this can be prohibitively large. Meanwhile if AD is restricted to forward mode, then the space required is much less, i.e., it is proportional to $\sigma(F)$, the space required to evaluate $F(x)$, and typically $\omega(F) \gg \sigma(F)$; however, forward mode alone can be much more costly than a combination of forward and reverse modes in terms of work required. For example, reverse mode can calculate the gradient of differentiable function $f : \mathbb{R}^n \mapsto \mathbb{R}^1$ in time proportional to $\omega(f)$ whereas forward mode requires $n \cdot \omega(f)$ operations. The following result formalizes the space and time requirements for the bi-coloring AD method[1] [15, 19].

Lemma 2.2.1. *Assume $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ is differentiable and its Jacobian matrix J is computed by the bi-coloring AD method [15]. Assuming the optimal coloring is found, then in general,*

$$\left.\begin{array}{l} \omega(J) = O(\chi_B(G^D(J)) \cdot \omega(F) + |J|_{NNZ}) \\ \sigma(J) = O(\omega(F) + |J|_{NNZ}) \end{array}\right\} \tag{2.1}$$

where $\chi_B(G^D(J))$ is the bi-coloring number[1] of J, and $|J|_{NNZ}$ is the number of nonzero entries in J.

Proof. According to [15], a bi-coloring for $J \in \mathbb{R}^{m\times n}$ corresponds to thin matrices $V \in \mathbb{R}^{n\times t_V}$ and $W \in \mathbb{R}^{m\times t_W}$, where J can be determined with work $O(|J|_{\mathrm{NNZ}})$ if $W^T J$ and JV are given. We can obtain J in $O(|J|_{\mathrm{NNZ}})$ because at least one nonzero entry is determined in one substitution, so at most $|J|_{\mathrm{NNZ}}$ substitutions are required. Now consider cost for calculating $W^T J$ and JV: The forward mode of AD allows for the computation of product JV in time proportional to $O(t_V \cdot \omega(F))$, and similarly reverse mode allows for the computation of product $W^T J$ in time proportional to $O(t_W \cdot \omega(F))$ [15]. If the optimal coloring is found

$$\chi_B(G^D(J)) = t_V + t_W,$$

and then

$$\begin{array}{rcl} \omega(J) & = & O((t_V + t_W) \cdot \omega(F) + |J|_{\mathrm{NNZ}}) \\ & = & O(\chi_B(G^D(J)) \cdot \omega(F) + |J|_{\mathrm{NNZ}}). \end{array}$$

The second equation in (2.1) is obviously true because the reverse mode of AD needs $O(\omega(F))$ space [15] and J itself needs $O(|J|_{\mathrm{NNZ}})$ space. $\square$

The bi-coloring AD method does not guarantee to find an optimal coloring, but heuristic coloring methods determine t_V, t_W aiming for $(t_V + t_W) \cong \chi_B(G^D(J))$. Therefore total work for computing J in practise is given by (2.1).

Consider now the (directed) computational graph that represents the structure of the program to evaluate $F(x)$:

$$\vec{G}(F) = (V, \vec{E}) \tag{2.2}$$

[1] Please refer to **Section** 4.2 for the details of the bi-coloring method and the bi-coloring number.

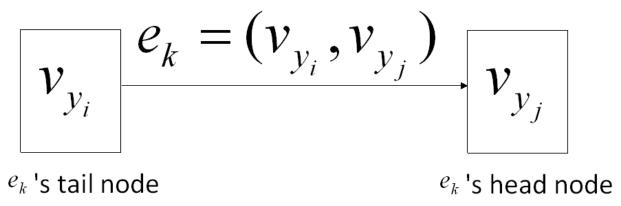


Figure 2.1: Head node and tail node of a given edge e_k

where V consists of three sets of vertices. Specifically, $V = \{V_x, V_y, V_z\}$ where vertices in V_x represent the input variables; a vertex in V_y represent ***both*** a basic or elementary *operation* receiving one or two inputs, producing a single ouput variable ***and*** the output *intermediate variable*; vertices in V_z represent the output variables. So input variable x_i corresponds to vertex $v_{x_i} \in V_x$, intermediate variable y_k corresponds to vertex $v_{y_k} \in V_y$, and output $z_j = [F(x)]_j$ corresponds to vertex $v_{z_j} \in V_z$. Note that the number of vertices in V_y, i.e., $|V_y|$, is the number of basic operations required to evaluate $F(x)$. Hence $\omega(F) = |V_y|$.

The edge set $\vec{E}$ represents the traffic pattern of the variables. For example, there is a directed edge $e_k = (v_{y_i}, v_{y_j}) \in \vec{E}$ if intermediate variable y_i is required by computational node v_{y_j} to produce intermediate variable y_j. If $e_k = (v_{y_i}, v_{y_j}) \in \vec{E}$ is a directed edge from vertex v_{y_i} to vertex v_{y_j} then we refer to vertex v_{y_i} as the *tail node* of edge e_k and vertex v_{y_j} as the *head node* of edge e_k. See **Figure** 2.1 for an illustration. It is clear that if F is well-defined then $\vec{G}(F)$ is an acyclic graph.

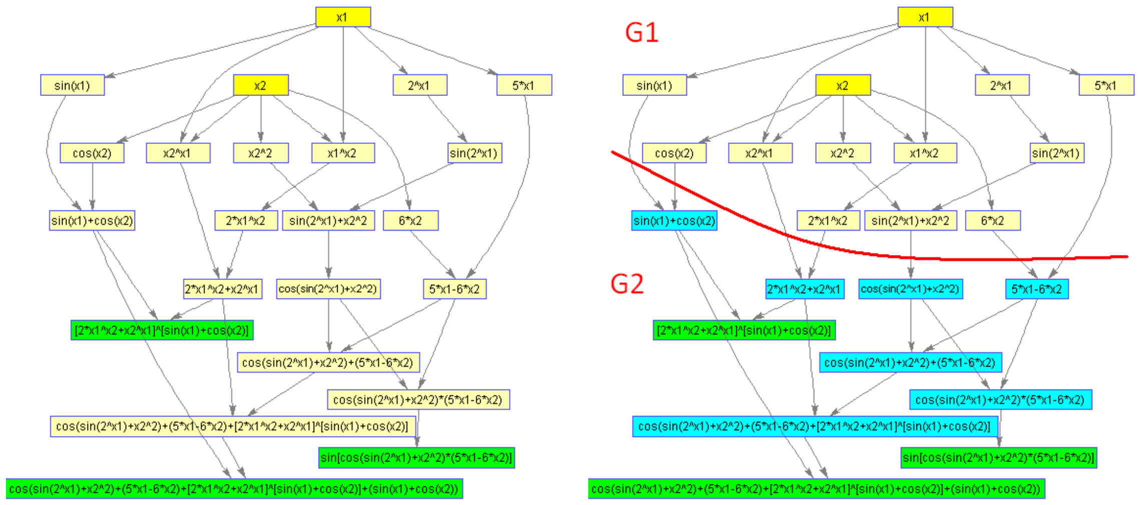
Example 2.2.2. *$F : \mathbb{R}^2 \mapsto \mathbb{R}^3$ is defined as:*

$$F\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} \sin(\cos(\sin 2^{x_1} + x_2^2) \cdot (5x_1 - 6x_2)) \\ (2x_1^{x_2} + x_2^{x_1})^{\sin x_1 + \cos x_2} \\ \cos(\sin 2^{x_1} + x_2^2) + (5x_1 - 6x_2) + (2x_1^{x_2} + x_2^{x_1}) + (\sin x_1 + \cos x_2) \end{bmatrix} \tag{2.3}$$

Then F 's computational graph is **Figure** 2.2(a).

Definition 2.2.3. *$E_d \subseteq \vec{E}$ is a* directed edge separator *in directed graph $\vec{G}$ if $\vec{G} - \{E_d\}$ consists of disjoint components $\vec{G}_1$ and $\vec{G}_2$ where all edges in E_d have the same orientation relative to $\vec{G}_1$, $\vec{G}_2$.*

Example 2.2.4. *One choice of an edge separator for F defined by equation (2.3) is given in **Figure** 2.2(b).*


G1
G2
x1
x2
sin(x1)
2^x1
5*x1
cos(x2)
x2^x1
x2^2
x1^x2
sin(2^x1)
sin(x1)+cos(x2)
2*x1^x2
sin(2^x1)+x2^2
6*x2
2*x1^x2+x2^x1
cos(sin(2^x1)+x2^2)
5*x1-6*x2
[2*x1^x2+x2^x1]^[sin(x1)+cos(x2)]
cos(sin(2^x1)+x2^2)+(5*x1-6*x2)
cos(sin(2^x1)+x2^2)*(5*x1-6*x2)
cos(sin(2^x1)+x2^2)+(5*x1-6*x2)+[2*x1^x2+x2^x1]^[sin(x1)+cos(x2)]
sin[cos(sin(2^x1)+x2^2)*(5*x1-6*x2)]
cos(sin(2^x1)+x2^2)+(5*x1-6*x2)+[2*x1^x2+x2^x1]^[sin(x1)+cos(x2)]+(sin(x1)+cos(x2))

(a) F's computational graph G

(b) An example of graph G's directed edge separator

Figure 2.2: An example of computational graphs and a sample directed separator

Suppose $E_d \subseteq \vec{E}_y$ is an edge separator of the computational graph $\vec{G}(F)$ with orientation forward in time. Then the nonlinear function $F(x)$ can be broken into two parts:

$$\left.\begin{array}{l}\text{solve for } y\text{: } F_1(x, y) = 0 \\ \text{solve for } z\text{: } F_2(x, y) - z = 0\end{array}\right\} \tag{2.4}$$

where y is the vector of intermediate variables defined by the *tail vertices* of the edge separator E_d, and z is the output vector, i.e., $z = F(x)$. Let p be the number of tail vertices of edge set E_d, i.e., $y \in \mathbb{R}^p$. Note: $|E_d| \geq p$. The nonlinear function F_1 is defined by the computational graph above E_d, i.e., G_1, and nonlinear function F_2 is defined by the computational graph below E_d, i.e., G_2. See **Figure** 2.2(b). We note that the system (2.2) can be differentiated wrt (x, y) to yield an 'extended' Jacobian matrix [14, 16, 26]:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & (F_1)_y \\ (F_2)_x & (F_2)_y \end{bmatrix} \tag{2.5}$$

Since y is a well-defined unique output of function $F_1 : \mathbb{R}^{n+p} \mapsto \mathbb{R}^p$, $(F_1)_y$ is a $p \times p$ nonsingular matrix. The Jacobian of F is the Schur-complement of (2.4), i.e.,

$$J(x) = (F_2)_x - (F_2)_y (F_1)_y^{-1} (F_1)_x. \tag{2.6}$$

There are two important computational issues to note. The first is that the work to evaluate J_E is often less than that required to evaluate $J(x)$ directly. The second is that less space is often required to calculate and save J_E relative to calculating and saving J directly by AD (when the AD technique involves the use of "reverse mode" as in the bi-coloring technique). **Theorem 2.2.5** formalizes this.

Theorem 2.2.5. *Assume the computational graph G is divided into two disjoint subgraphs G_1, G_2 with the removal of directed edge separator E_d as described above. Let J_E be computed by the bi-coloring technique [15]. Then assuming optimal graph coloring, in general,*

$$\left.\begin{array}{rcl} \omega(J_E) & = & O(\chi_B(G^D[(F_1)_x,(F_1)_y])\cdot\omega(F_1)+\chi_B(G^D[(F_2)_x,(F_2)_y])\cdot\omega(F_2)+|J_E|_{NNZ}) \\ \sigma(J_E) & = & O(\max(\sigma[(F_1)_x,(F_1)_y],\sigma[(F_2)_x,(F_2)_y])+|J_E|_{NNZ}) \\ & = & O(\max(\omega(F_1),\omega(F_2))+|J_E|_{NNZ}) \end{array}\right\} \tag{2.7}$$

Proof. With reference to (2.5), we first determine $\begin{bmatrix}(F_1)_x & (F_1)_y\end{bmatrix}$, and then determine $\begin{bmatrix}(F_2)_x & (F_2)_y\end{bmatrix}$. By **Lemma 2.2.1**,

$$\begin{array}{rcl} \omega(J_E) & = & O(\chi_B(G^D[(F_1)_x,(F_1)_y])\cdot\omega(F_1)+|[(F_1)_x,(F_1)_y]|_{\text{NNZ}} \\ & + & \chi_B(G^D[(F_2)_x,(F_2)_y])\cdot\omega(F_2)+|[(F_2)_x,(F_2)_y]|_{\text{NNZ}}) \\ & = & O(\chi_B(G^D[(F_1)_x,(F_1)_y])\cdot\omega(F_1)+\chi_B(G^D[(F_2)_x,(F_2)_y])\cdot\omega(F_2)+|J_E|_{\text{NNZ}}). \end{array}$$

Now consider the space: To determine $\begin{bmatrix}(F_1)_x & (F_1)_y\end{bmatrix}$, $O(\omega(F_1))$ is required for the reverse mode of AD. Then to evaluate $\begin{bmatrix}(F_2)_x & (F_2)_y\end{bmatrix}$, previous memory can be cleared and needs $O(\omega(F_2))$ space for F_2's reverse mode. Hence total space requirement is the peak usage which is $O(\max(\omega(F_1),\omega(F_2)))$. Extra $O(|J_E|_{\text{NNZ}})$ space is to restore results obtained, and hence

$$\sigma(J_E)=O(\max(\omega(F_1),\omega(F_2))+|J_E|_{\text{NNZ}}).$$

□

We can compare (2.7) to (2.1) to contrast the time/space requirements of the separator/bi-coloring AD approach to obtain J_E versus the time/space requirements of the bi-coloring AD method to obtain J. Specifically, and most importantly, the space requirement typically decreases: (2.7, $\sigma(J_E)$) indicates that the required space is $O(\max(\omega(F_1),\omega(F_2)))$, since the term $|J_E|_{NNZ}$ is usually dominated by the first, whereas determining J directly by bi-coloring takes space approximately proportional to $O(\omega(F))$, by (2.1). For example, if the edge separator divides $G(F)$ into two equal-sized pieces G_1, G_2, then $\sigma(J_E) \simeq \omega(F)/2 \simeq \sigma(J)/2$; that is, we have essentially halved the space requirements.

The computational cost, comparing (2.7,$\omega(J_E)$) to (2.1,$\omega(J)$) can either increase or decrease. However, due to increased sparsity usually

$$\begin{array}{rcl} \chi_B(G^D[(F_1)_x,(F_1)_y]) & \leqslant & \chi_B(G^D(J)) \\ \chi_B(G^D[(F_2)_x,(F_2)_y]) & \leqslant & \chi_B(G^D(J)), \end{array} \tag{2.8}$$

and then

$$\begin{array}{rl} & \chi_B(G^D[(F_1)_x,(F_1)_y]) \cdot \omega(F_1) + \chi_B(G^D[(F_2)_x,(F_2)_y]) \cdot \omega(F_2) \\ \leqslant & \chi_B(G^D) \cdot (\omega(F_1) + \omega(F_2)) \\ = & \chi_B(G^D) \cdot \omega(F), \end{array} \tag{2.9}$$

and so in this case there is a no increase (and typically a reduction) in computational cost. The upshot is that use of the edge separator often results in cost savings both in time and in space (when computing J_E rather than J).

It is usually less expensive, in time and space, to compute $J_E(x)$ rather than $J(x)$, using a combination of forward and reverse modes of automatic differentiation[18]. However, what is the utility of $J_E(x)$? The answer is that $J_E(x)$ can often be used directly to simulate the action of J and this computation can often be less expensive (due to sparsity in J_E that is not present in J) than explicitly forming and using J. For example, the Newton system 'solve $Js = -F$' can be replaced with

$$\text{solve } J_E \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} 0 \\ -F \end{bmatrix}. \tag{2.10}$$

The main points are that calculating matrix J_E can be less costly than calculating matrix J, and solving (2.10) can also be relatively inexpensive given sparsity that can occur in J_E that may not be present in J.

2.3 Automatic Differentiation and Multiple Edge Separators

The ideas discussed above can be generalized to the case with multiple mutually independent directed edge separators, $E_{d_1}, \cdots, E_{d_k} \in \vec{E}$, where we assume $G - \{E_{d_1}, \cdots, E_{d_k}\} = \{G_1, \cdots, G_{k+1}\}$. The connected graphs $G_1, \cdots, G_{k+1}$ are pairwise disjoint and are ordered such that when evaluating F, G_i can be fully evaluated before $G_{i+1}, i = 1, \cdots, k$.

Suppose $E_{d_1}, \cdots, E_{d_k} \in \vec{E}$ are pairwise disjoint separators of the computational graph $\vec{G}(F)$ with orientation forward in time (as indicated above). Then the evaluation of nonlinear function $F(x)$ can be broken into $k+1$ steps:

$$\left.\begin{array}{lcl} \text{solve for } y_1 & : & F_1(x, y_1) = 0 \\ \text{solve for } y_2 & : & F_2(x, y_1, y_2) = 0 \\ \quad\vdots & & \qquad\vdots \\ \text{solve for } y_k & : & F_k(x, y_1, \cdots, y_k) = 0 \\ \text{solve for } z & : & F_{k+1}(x, y_1, \cdots, y_k) - z = 0 \end{array}\right\} \tag{2.11}$$

where y_i is the vector of intermediate variables defined by the *tail vertices* of the edge separator E_{d_i}, for $i = 1, \cdots, k$ and z is the output vector, i.e., $z = F(x)$. Let p_i be the number of tail vertices of edge set E_{d_i}, i.e., $y_i \in \mathbb{R}^{p_i}$. The nonlinear function F_i is defined by the computational graph to the left of E_{d_i}, i.e., G_i. We note that the system (2.11) can be differentiated wrt (x, y) to yield an 'extended' Jacobian matrix:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & (F_1)_{y_1} & 0 & 0 & 0 & 0 \\ (F_2)_x & (F_2)_{y_1} & (F_2)_{y_2} & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & . & \vdots \\ (F_k)_x & (F_k)_{y_1} & (F_k)_{y_2} & \cdots & \cdots & (F_k)_{y_k} \\ (F_{k+1})_x & (F_{k+1})_{y_1} & (F_{k+1})_{y_2} & \cdots & \cdots & (F_{k+1})_{y_k} \end{bmatrix} \tag{2.12}$$

We note that matrix J_E is a block lower-Hessenberg matrix; moreover, since all intermediate variables are well-defined for arbitrary input vectors it follows that the super-diagonal blocks $(F_1)_{y_1}, (F_2)_{y_2}, \cdots, (F_k)_{y_k}$ are all non-singular; e.g., matrix $(F_i)_{y_i}$ is a $p_i \times p_i$ nonsingular matrix where p_i is the length of vector y_i. The extended Jacobian matrix is of dimension $(m + \sum_{i=1}^{k} p_i) \times (n + \sum_{i=1}^{k} p_i)$.

In analogy to the 1-separator case, we argue below that the matrix J_E can often be calculated more efficiently than the Jacobian of $F(x)$, i.e., $J(x)$. In addition, due to the increased sparsity/structure in J_E, the Newton system 'solve $Js = -F$' can often be solved more efficiently by solving

$$J_E \begin{bmatrix} s \\ t_1 \\ \vdots \\ t_k \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ -F \end{bmatrix}. \tag{2.13}$$

We note that again a Schur-complement computation can yield the Jacobian matrix J given the extended Jacobian J_E. Specifically, if we define:

$$\begin{aligned}
A &= [(F_1)_x, (F_2)_x, \cdots, (F_k)_x]^T \\
B &= \begin{bmatrix} (F_1)_{y_1} & 0 & 0 & \cdots & 0 \\ (F_2)_{y_1} & (F_2)_{y_2} & 0 & \cdots & 0 \\ (F_3)_{y_1} & (F_3)_{y_2} & (F_3)_{y_3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (F_k)_{y_1} & (F_k)_{y_2} & (F_k)_{y_3} & \cdots & (F_k)_{y_k} \end{bmatrix} \\
C &= [F_{k+1}]_x \\
D &= [(F_{k+1})_{y_1}, (F_{k+1})_{y_2}, (F_{k+1})_{y_3}, \cdots, (F_{k+1})_{y_k}],
\end{aligned}$$

then

$$J_E = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \tag{2.14}$$

where B is nonsingular, and

$$J = C - DB^{-1}A. \tag{2.15}$$

The space/time requirements for the multi-separator case will be formalized in **Theorem 2.3.1**, in analogy to the 1-separator case captured by **Theorem 2.2.5**.

Theorem 2.3.1. *Assume the computational graph G is divided into $k+1$ disjoint subgraphs $G_1, G_2, \cdots, G_{k+1}$ with the removal of directed edge separators $E_{d_1}, E_{d_2}, \cdots, E_{d_k}$ as described above. Let J_E be computed by the bi-coloring technique [15]. Then assuming optimal graph coloring, in general,*

$$\left.\begin{aligned}
\omega(J_E) &= O(\chi_B(G^D[(F_1)_x, (F_1)_{y_1}]) \cdot \omega(F_1) \\
&\quad + \chi_B(G^D[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}]) \cdot \omega(F_2) \\
&\quad + \cdots \\
&\quad + \chi_B(G^D[(F_k)_x, (F_k)_{y_1}, \cdots, (F_k)_{y_k}]) \cdot \omega(F_k) \\
&\quad + \chi_B(G^D[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]) \cdot \omega(F_{k+1}) \\
&\quad + |J_E|_{NNZ}) \\
&= O(\textstyle\sum_{i=1}^{k}(\chi_B(G^D[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}]) \cdot \omega(F_i)) \\
&\quad + \chi_B(G^D[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]) \cdot \omega(F_{k+1}) \\
&\quad + |J_E|_{NNZ})) \\
&= O(\textstyle\sum_{i=1}^{k+1}(\chi_B(G^D[F_i]) \cdot \omega(F_i)) + |J_E|_{NNZ})) \\
\sigma(J_E) &= O(\max(\sigma[(F_1)_x, (F_1)_{y_1}], \sigma[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}], \cdots, \\
&\quad \sigma[(F_k)_x, (F_k)_{y_1}, \cdots, (F_k)_{y_k}], \sigma[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]) + |J_E|_{NNZ}) \\
&= O(\max_{i=1,\cdots,k+1}(\omega(F_i)) + |J_E|_{NNZ})
\end{aligned}\right\} \tag{2.16}$$

Proof. With reference to (2.12), to determine J_E, we determine $[(F_1)_x, (F_1)_{y_1}]$, $[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}]$, $\cdots$, $[(F_k)_x, (F_k)_{y_1}, \cdots, (F_k)_{y_k}]$, and $[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]$ sequentially. By **Lemma 2.2.1**,

$$\begin{aligned}
\omega(J_E) &= O(\chi_B(G^D[(F_1)_x, (F_1)_{y_1}]) \cdot \omega(F_1) + |[(F_1)_x, (F_1)_{y_1}]|_{\text{NNZ}} \\
&\quad +\chi_B(G^D[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}]) \cdot \omega(F_2) + |[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}]|_{\text{NNZ}} \\
&\quad +\cdots \\
&\quad +\chi_B(G^D[(F_k)_x, (F_k)_{y_1}, \cdots, (F_k)_{y_k}]) \cdot \omega(F_k) + |[(F_k)_x, (F_k)_{y_1}, \cdots, (F_k)_{y_k}]|_{\text{NNZ}} \\
&\quad +\chi_B(G^D[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]) \cdot \omega(F_{k+1}) + |[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]|_{\text{NNZ}} \\
&= O(\textstyle\sum_{i=1}^{k}(\chi_B(G^D[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}]) \cdot \omega(F_i)) \\
&\quad +\chi_B(G^D[(F_{k+1})_x, (F_{k+1})_{y_1}, \cdots, (F_{k+1})_{y_k}]) \cdot \omega(F_{k+1}) \\
&\quad +|J_E|_{\text{NNZ}})) \\
&= O(\textstyle\sum_{i=1}^{k+1}(\chi_B(G^D[F_i]) \cdot \omega(F_i)) + |J_E|_{\text{NNZ}})).
\end{aligned}$$

Now consider the space: To determine $[(F_1)_x, (F_1)_{y_1}]$, $O(\omega(F_1))$ space is required for the reverse mode of AD. Then to evaluate $[(F_2)_x, (F_2)_{y_1}, (F_2)_{y_2}]$, previous memory can be cleared and needs $O(\omega(F_2))$ space for F_2's reverse mode. Similarly for arbitary $i = 1, \cdots, k{+}1$, $O(\omega(F_i))$ space is required for F_i's reverse mode. Hence total space requirement is the peak usage which is $O\left(\max_{i=1:k+1}(\omega(F_i))\right)$. Extra $O(|J_E|_{\text{NNZ}})$ space is to restore results obtained, and hence

$$\sigma(J_E) = O(\max_{i=1,\cdots,k+1}(\omega(F_i)) + |J_E|_{\text{NNZ}}).$$

□

Note that if the edge separators divide $\vec{G}(F)$ into equal-size pieces $\vec{G}_1, \vec{G}_2, \cdots, \vec{G}_{k+1}$, the space requirement can be simplified to

$$\sigma(J_E) = \frac{\omega(F)}{k+1} + |J_E|_{\text{NNZ}}\,. \tag{2.17}$$

Given any number of k edge separators, the first term in equation (2.17) decreases as k increases. On the other hand, the size of extended Jacobian matrix J_E increases as number of intermediate variables y increases. It is clear that the optimal value of k to minimize the space requirement is a balance between the two terms in equation (2.17).

It is helpful to consider the two extreme cases. If no edge separator is chosen, then $J_E = J$ hence size of J_E remains $m \times n$. However the first term will be $\omega(F)$, which can be large. In the other extreme case with finest granularity, the first term in equation (2.17) is negligible but the second term is proportional to $\omega(F)$. Determining the optimal value k is a combinatorial optimization problem, which is hard to solve. It is useful to build up models to study the dependency on k and other behaviours for this problem.

The number of nonzeros in J_E, $|J_E|_{NNZ}$, is problem dependent. We can first assume that there is a constant β such that the number of nonzeros in each row of J_E is bounded by β. The number of rows of J_E is problem dependent too. We will assume that the number of row is $m + p(k)$, where m is the dimension of input variable x, and $p(k)$ is the total dimension of intermediate variables $y_1, y_2, \cdots, y_k$. We further assume that the average size of intermediate variables is p_{avg}, which is independent of the number of separators k. In this case the number of rows in J_E is $m + p(k) = m + k \cdot p_{avg}$. The space requirement in equation (2.17) can now be simplified to the following inequality:

$$\sigma(J_E) = \frac{\omega(F)}{k+1} + |J_E|_{\text{NNZ}} \leq \frac{\omega(F)}{k+1} + (m + k \cdot p_{avg}) \cdot \beta, \tag{2.18}$$

where the optimal value of occurs at

$$k^* = \sqrt{\frac{\omega(F)}{p_{avg} \cdot \beta}} - 1,$$

at which the space requirement upper bound is

$$\sigma(J_E^{k^*}) \leq 2\sqrt{\omega(F) \cdot p_{avg} \cdot \beta} + (m-1) \cdot \beta. \tag{2.19}$$

The result in inequality (2.19) relies on several assumptions, however it is still very instructive. Compared to the first term in inequality (2.19), the second term is neglectable. If we only consider the first term $2\sqrt{\omega(F) \cdot p_{avg} \cdot \beta}$, the space requirement is the square root of that required by traditional AD. Moreover, the factor $\sqrt{p_{avg}}$ suggests that smaller edges separators are greatly preferable.

Chapter 3

On Finding Edge Separators to Increase Efficiency in the Application of Automatic Differentiation

3.1 Introduction

In **Section** 2.2 we observed that if a small directed edge separator divides the computational graph $\vec{G}$ into roughly two equal components $\vec{G}_1$ and $\vec{G}_2$, then the space requirement are minimized (roughly halved). Moreover, the required work, as indicated by equation (2.7), will not increase, and due to increased sparsity, will likely decrease.

Therefore, our approach is to seek a small directed edge separator that will (roughly) bisect the fundamental computational graph. In this section, we present two algorithms to find good separators.

3.2 Weighted Minimum Separator

This minimum weighted separator approach is based on the Ford-Fulkerson (FF) algorithm [22], a well known max-flow/min-cut algorithm. The Ford-Fulkerson algorithm finds the minimum $s-t$ cut, a set of edges whose removal separates specified node s and node t, two arbitrary nodes in the graph. A minimum cut does not always correspond to a directed separator, hence we "post process" the min-cut solution to obtain a directed separator.

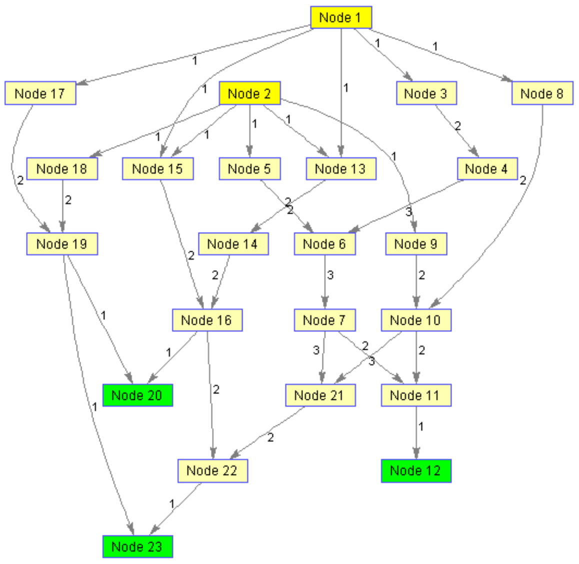


Figure 3.1: Depth of edges in F's (equation (2.3)) computational graph

We desire that the determined separator (roughly) divides the fundamental computational graph in half. To add this preference into the optimization, we assign capacities to edges to reflect distance from the input or output nodes, whichever is closer. With this kind of weight distribution, a 'small' separator will likely be located towards the middle of the fundamental computational graph.

To determine the weights we first calculate depth of nodes and edges.

Definition 3.2.1. *We define the* depth *of a node v in a DAG to be the shorter of shortest directed path from an input node (source) to v and the shortest directed path from v to an output node (sink). We define the* depth *of an edge y in a DAG in an analogous fashion.*

Example 3.2.2. ***Figure*** *3.1 is depth of edges in F's computational graph, where F is defined by equation (2.3).*

Define a decreasing function

$$f : \mathbb{Z}^+ \mapsto \mathbb{Z}^+ \tag{3.1}$$

taking depth as inputs and gives weights for each edge as outputs. Notice weights are restricted to be integers, because the original Ford-Fulkerson algorithm assumes integral edge weights and has running time bounded in terms of the final flow. Notice that the

Edmonds-Karp variant of the FF algorithm allows arbitrary real weights and runs in polynomial time regardless of the magnitude of the flow. For the Edmonds-Karp algorithm, the running time is determined by the size of the network. However, for simplicity, in the numerical experiments we use the original Ford-Fulkerson algorithm to find the min-cuts for the weighted graphs.

There are many possible choices for f. Currently we find that quadratic weights give reasonable results. See **Chapter** 3.5 for numerical examples. Once weights are determined, the Ford-Fulkerson algorithm can be applied and edge separators can be obtained.

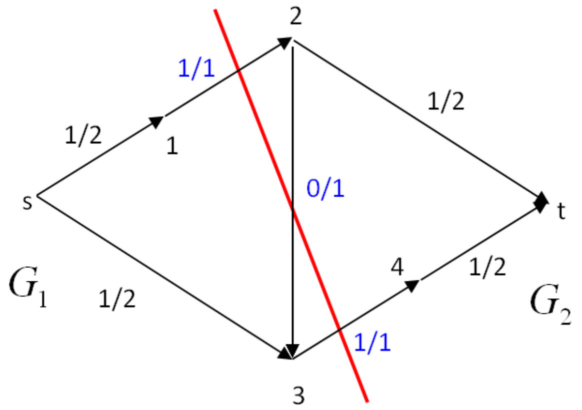
Hence our proposed method is as follows:

1. Assign weights to edges to reflect depth of an edge;
2. Solve the weighted mincut problem, e.g. using the Ford-Fulkerson method;
3. If the cut is not a directed separator, modify according to **Algorithm** 3.2.3.

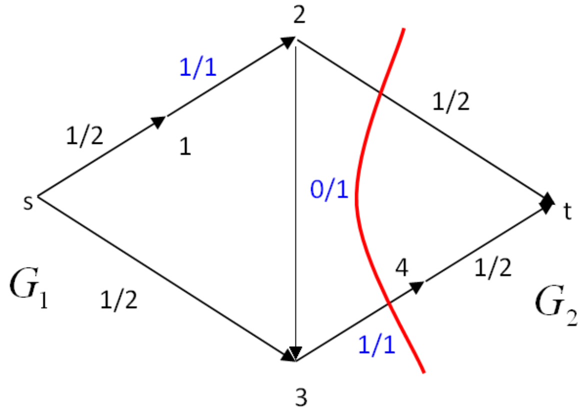
Algorithm 3.2.3. *Let $E_u \subseteq \vec{E}$ such that graph $\vec{G} - E_u$ consists of two components $\vec{G}_1$ and $\vec{G}_2$, where source nodes are in $\vec{G}_1$ and sink nodes are in $\vec{G}_2$. If E_u is not a directed separator, then E_u contains both edges from $\vec{G}_2$ to $\vec{G}_1$ and edges from $\vec{G}_1$ to $\vec{G}_2$. Let $S = V(\vec{G}_1)$ and $T = V(\vec{G}_2)$. A directed separator $E_d = \vec{E}(S, T)$ can be generated either by moving tail nodes of $T \to S$ edges from T to S recusively, or by moving head nodes of $T \to S$ edges from S to T recursively. The formal description is stated as follows:*

1. $T_1 \leftarrow \{v : v \in T\} \cup \{v :$ *there exists a directed uv-path in* $\vec{G}, u \in T\}$;
2. $S_1 \leftarrow V(\vec{G}) - T_1, E_1 = E(\vec{G}) - E(\vec{G}(S_1)) - E(\vec{G}(T_1))$;
3. $S_2 \leftarrow \{v : v \in S\} \cup \{v :$ *there exists a directed vu-path in* $\vec{G}, u \in S\}$;
4. $T_2 \leftarrow V(\vec{G}) - S_2, E_2 = E(\vec{G}) - E(\vec{G}(S_2)) - E(\vec{G}(T_2))$;
5. *Pick the smaller between E_1 and E_2 as the desired separator.*

Compared with edge separators with no direction constraint, finding a directed separator is usually harder. Considering this, we sometimes use separators locating schemes which do not guarantee directedness. In this case the above algorithm can always be used to 'direct' such an edge separator. The following is an example for directing an edge separator.



(a) A constructed undirected edge separator



(b) Edge separator becomes directed after adjustment

Figure 3.2: An example of an undirected edge separator and corresponding adjustment

Cost Analysis

For integral weights, the run time of Ford-Fulkerson is bounded by $O(|E|f)$ [22] where $|E|$ is number of edges in the graph, and f is the maximum flow. In computational graphs, equation (3.1) is always a decreasing function so weights in the middle are smaller. Due to this special distribution, usually there exists a edge separator in the middle with a small total weight, bounding f. In practice, in most cases $O(|E|f)$ is just a very loose upper bound and running time is more like $O(|E|)$.

Example 3.2.4. *Use function defined in equation (3.5) to test. Notice that F_k gets more complicated as k increases. The time used to evaluate F_k and find cutsets is shown in **Figure** 3.3.*

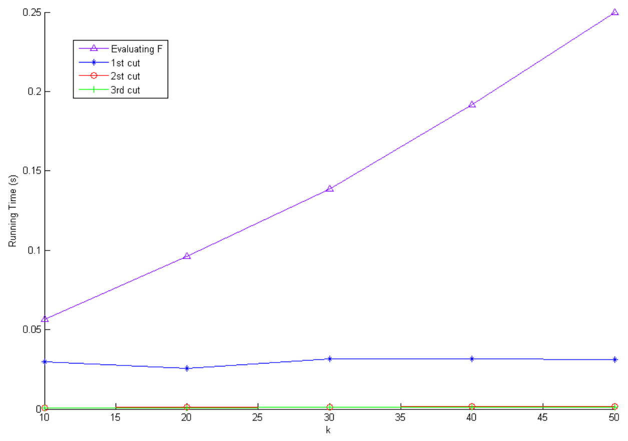


Figure 3.3: Performance of Ford-Fulkerson algorithm on F_k

3.3 Natural Order Edge Separator

A second method to generate directed separators comes from the observation that if the 'tape' generated by reverse-mode AD is snipped at any point then effectively a directed separator is located.

As described in (2.2), computational graphs represent the process of evalution of functions. They are generated based on information recorded during the execution of the reverse mode. These records are called computational tapes, each of which contains a long vector of cells. Each cell represents a basic operation of the input function, hence corresponds to a node in the computational graph. Cells in a tape are ordered according to the execution time of its corresponding basic operation, therefore cells are in chronological order. Permutation on tape method is based on this property of computational tapes.

Suppose we are given a computational graph $\vec{G}$ and the correponding computational tape T with length $|V(\vec{G})|$. A natural partition $(\vec{G}_1,\vec{G}_2)$ of $\vec{G}$ is $\vec{G}_1 = \vec{G}(T(1 : i))$, $\vec{G}_2 = \vec{G}(T(i+1 : |V(\vec{G})|))$, where i is some integer between 1 and $|V(\vec{G})|-1$. Since cells in the tape are in chronological order, all basic operations represented in $\vec{G}_1$ are evaluated before those represented in $\vec{G}_2$, therefore all edges between $\vec{G}_1$, $\vec{G}_2$ are directed from $\vec{G}_1$ to $\vec{G}_2$. Since these edges form a directed edge separator, we can then choose i to get the preferred edge separator in terms of separator size and partition ratio. The following is a formal description to global version natural order edge separator method.

Algorithm 3.3.1. *Suppose a cost function f^{cost} is defined on an directed edge separator E_d, representing the quality of E_d in terms of separator size and partition ratio, and the function value decreases as the quality increases. Then the following are the steps finding the optimal natural order edge separator.*

1. *Perform forward sweep on candidate function F, and generate the tape T for F;*
2. $i \leftarrow 1, i^* \leftarrow 0, f^* \leftarrow +\infty, E_d^* \leftarrow \emptyset$;
3. $E_{d_i} \leftarrow E(T(1:i), T(i+1:|V(\vec{G})|))$;
4. *if $f^{cost}(E_{d_i}) < f^*$, then $i^* \leftarrow i$, $E_d^* \leftarrow E_{d_i}$, and $f^* \leftarrow f^{cost}(E_{d_i})$;*
5. *if $i < |V(\vec{G})| - 1$, then $i \leftarrow i+1$ and return to step 3, otherwise proceed to step 6;*
6. *E_d^* is the optimal natural order edge separator.*

Notice that there are many choices to the cost function f^{cost}. In numerical experiments the f^{cost} we use is the sparsity function defined by

$$f^{cost} = \frac{|E(\vec{G}_1, \vec{G}_2)|}{\min(|\vec{G}_1|, |\vec{G}_2|)}. \tag{3.2}$$

The good choice of the cost function is a topic for future research.

3.4 Multiple Edge Separators

Either of the proposed directed separator methods can be applied, recursively, to yield multiple separators. We do exactly this in our code and in our computational experiments below, always working on the largest remaining subgraph:

Algorithm 3.4.1. *If we are given a partition $V_1, V_2, \cdots, V_i$ of $V(\vec{G})$, where $\forall a, b = 1, \cdots, i, a \neq b$, edges between $\vec{G}(V_a)$ and $\vec{G}(V_b)$ are in one direction. (This partition defines $i-1$ directed separators.)*

1. $j \leftarrow \arg\max_{k=1,\cdots,i} |V_k|$;
2. *$E_j \leftarrow$ the directed separator in $\vec{G}(V_j)$ founded by either of the methods. Suppose $\vec{G}(V_j) - E_j$ consists of S_j and T_j;*

3. *$V_j \leftarrow V(S_j)$ and $V_{i+1} \leftarrow (T_j)$;*

4. *Repeat above step 1-3 until enough directed separators are generated.*

3.5 Experiments

In this section we provide computational results on some preliminary experiments to automatically reveal a 'AD-friendly' structure using the separator idea. These experiments are based on the minimum weighted separator algorithm and natural order separator algorithm described in the previous section, to find directed edge separators that bisect the fundamental computational graph.

In particular, in weighted minimum edges separator approach, we use two weighing strategies, quadratic weighing and 1-∞ weighing.

Definition 3.5.1. *The* quadratic weighing *is based on the fundamental computational graph, defined by*

$$W_{e_i} = (\max_j D_{e_j} - D_{e_i})^2 + 1, \tag{3.3}$$

*where depth D_e is calculated by **Definition** 3.2.1.*

Definition 3.5.2. *The* 1-∞ weighing *is also based on the fundamental computational graph, defined as follows:*

1. *$S \leftarrow \varnothing; T \leftarrow \varnothing$;*

2. *Calculate depth for each node in the computational graph by **Definition** 3.2.1;*

3. *$S \leftarrow S \cup \{s\}; T \leftarrow T \cup \{t\}$; where in graph $\vec{G} - S - T$, s is a source node with smallest depth and t is a sink node with smallest depth (the depth calculated in step 2);*

4. *Repeat previous step until $|S| = |T| \geqslant |V(\vec{G})|/4$;*

5. *If both ends of an edge is in $\vec{G} - S - T$, assign 1 to it as its weight, otherwise assign infinity to it.*

The cutset found using 1-∞ weighting scheme is equivalent to the minimum-cut in $\vec{G} - S - T$. In the case that S and T are adjacent or even overlapping in $\vec{G}$, we can similarly find another minimum-cut for $\vec{G}(S \cup T)$, and then combine it with the one in

$\vec{G} - S - T$, to get a directed edge separator in $\vec{G}$. A directed edge separator can then be generated from this edge cutset by algorihtm described in **Algorithm** 3.2.3.

We use the AD-tool, ADMAT [31], to generate the computational graphs. However, for efficiency reasons, ADMAT sometimes condenses the fundamental computational graph to produce a condensed computational graph. In a condensed computatonal graph nodes may represent matrix operations such as matrix-multiplication. Therefore our weighting heuristic is adjusted to account for this.

In our numerical experiments we focus on two types of structures that represent the two shape extreme cases.

3.5.1 Thin Computational Graphs

A function involving recusive iterations usually produces a "thin" computational graph.

Example. Define

$$F\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_3 \cdot \cos(\sin(2^{x_1} + x_2^2)) \\ 5x_1 - 6x_2 \\ 2x_2^{x_2} + x_2^{x_1} \end{bmatrix}, \tag{3.4}$$

and

$$F_1 = F \circ F \circ F \circ F \circ F \circ F.$$

Note that F_1's computational graph is long and narrow (i.e. 'thin').

After three iterations, three separators in **Figure** 3.4 are found. The graph is divided into four subgraphs. Visually, these edge separators are good in terms of size and evenly dividing the graph.

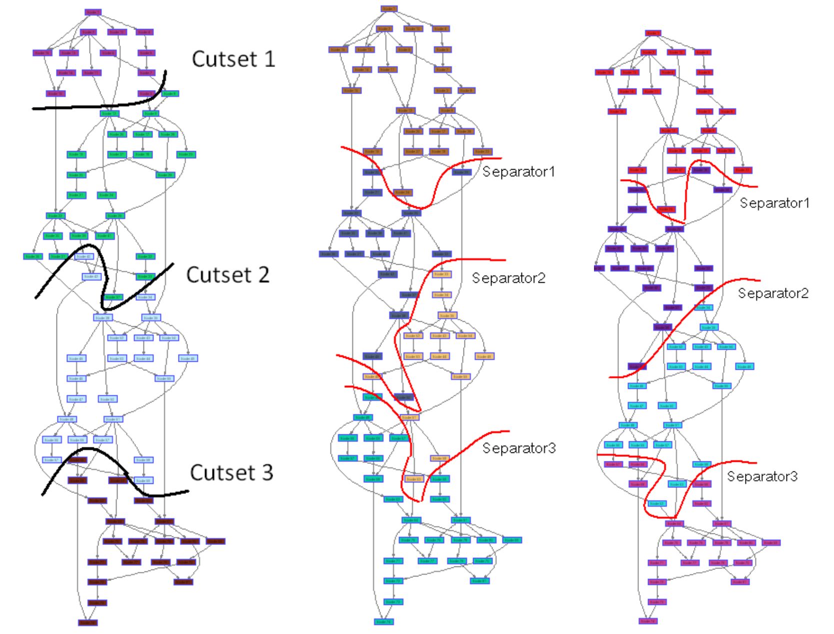
3.5.2 Fat Computational Graphs

A "fat" computational graph is produced when macro-computations are independent of each other. A typical example is:

$$F_2 = \sum_{i=1}^{6} F(x + \text{rand}_i(3,1)),$$

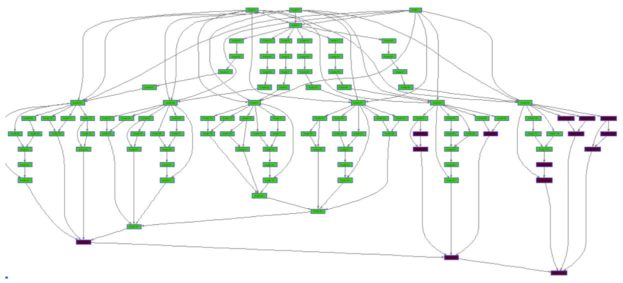
where F is defined by equation (3.4) in the previous experiment.

The separators found by our two algorithms on this example are useful but are less than ideal in contrast to the separators found in the "long thin" class. This is a topic for future experimental research.

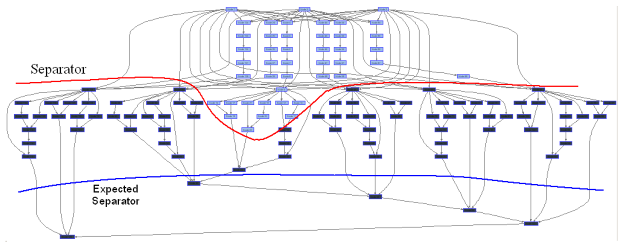


(a) Qudratic Minimum Weighted Separator (b) 1-∞ Minimum Weighted Separator (c) Natural Order Separator

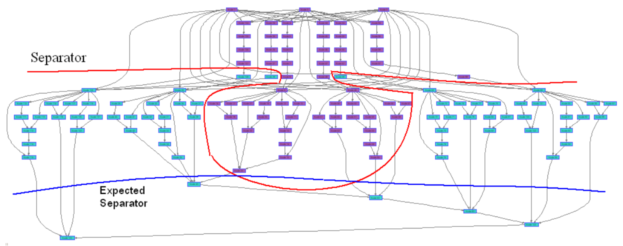
Figure 3.4: Obtained separators of F_1's condensed computational graph by the two different algorithms



(a) Quadratic Minimum Weighted Separator



(b) 1-∞ Minimum Weighted Separator



(c) Natural Order Separator

Figure 3.5: Obtained separators of F_2's condensed computational graph by the two different algorithms

3.6 Accelerating the Calculation of the Jacobian matrix

To illustrate how directed edge separators accelerate computation, we construct the following numeric example:

Let

$$f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{x_2+3x_3}{4} \\ \sqrt{x_1 x_3} \\ \frac{x_1+2x_2+x_3}{4} \end{bmatrix},$$

and

$$F_k = \underbrace{f \circ f \circ \cdots \circ f}_{k\ f\text{'s}}. \tag{3.5}$$

It is obvious that $F_n \equiv F_{k_1} \circ F_{k_2} \circ \cdots \circ F_{k_m}$ provided $n = \sum_{i=1}^{m} k_i$.

Now we try to calculate the Jacobian matrix $J \in \mathbb{R}^{3\times 3}$ of $F_{2400}(x_0)$ at $x_0 = [6, 9, 3]^T$. We will use ADMAT[31] reverse mode to obtain J both directly and by constructing edge separators. Their running time and space usage will be recorded to see improvements. If cutset method is used, J_E defined by equation (2.12) will be calculated, and J will be further calculated by equation (2.14) and (2.15).

Figure 3.6 illustrates our results.

For cases that directed separators method is introduced, computational graph is always divided evenly. For example: In one separator case, F_{2400} is treated as $F_{1200} \circ F_{1200}$. In two separators case, F_{2400} is treated as $F_{800} \circ F_{800} \circ F_{800}$, etc.. Space and time requirement decreases inversely as number of cuts increases, which is the same as the prediction of **Theorem 2.2.5**. This is a remarkable result.

The performance plot in **Figure** 3.6 did not count in time used to locate directed edge separators. The 'running time' refers to the time used to obtain Jacobian matrix with separators provided. In practice, generation of computational graphs and analysis may be costly, compared with those used to get Jacobian matrix directly. However once the desired edge separators are located, they are likely to be reusable in computations with same/similar functions but different initial input values. Therefore this optimization is useful in terms of the long run. For example if one want to calculate $F_{2400}(x_0)$'s Jacobian matrix many times at different points, x_0, then separator method need only be applied a single time.

The computational graph of F_k is a long thin graph. Our method locates small separators that tend to break the graph in a well-balanced way. So edge separators optimization

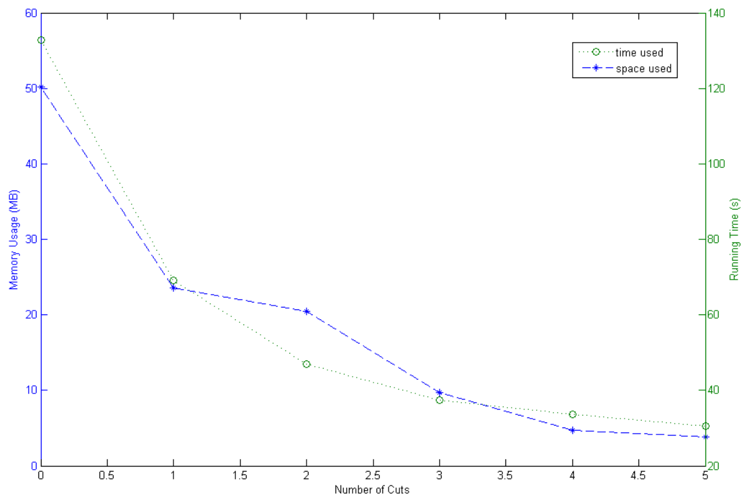


Figure 3.6: Acceleration of directed edge separators method

is expected to have good performance. In practice, the computational graphs are not so ideal; hence running time and memory may not be reduced so dramatically, but we expect a significant improvement.

Chapter 4

A Combination of Sparsity Techniques and Edge Separator Method

4.1 Introduction

In many scientific computing applications, the functions have hidden structure, i.e., they are mostly involved with sparse operations meanwhile having dense Jacobian matrices. For such functions, since their Jacobian matrices are dense, the bi-coloring methods[15] and other sparsity techniques cannot improve the efficiency of automatic differentiation. However, the edge separator method is able to reveal the hidden structure, and make the sparsity techniques applicable again. Let us consider the following two examples:

Example 4.1.1. *Define $F : \mathbb{R}^n \mapsto \mathbb{R}^m$:*

$$F(x) = A(x)^{-1}F_{sparse}(x)$$

where $A(x) \in \mathbb{R}^{m\times m}$ is a nonsingular matrix depending on x, and $F_{sparse} : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a sparse function, whose computational cost represents the major portion for the overall computation, i.e., $w(F_{sparse}(\cdot)) \gg w(A(x)^{-1}(\cdot))$.

Because of the inverse operation $A(x)^{-1}(\cdot)$, the Jacobian matrix of F in general is dense. Therefore sparsity techniques do not accelerate the determination of the Jacobian matrix and the computing time is proportional to $\min(m,n)\cdot\omega(F)$. Now suppose we apply the directed edge separator method on F first and generated $p+1$ sub-functions $F_1, F_2, \cdots, F_{p+1}$, *then F_{p+1} is the only sub-function containing the 'dense operation' $A(x)^{-1}(\cdot)$. Sparsity techniques now can be applied to accelerate all other sub-functions $F_1, F_2, \cdots, F_p$. According to **Theorem** 2.3.1, the computing time now is proportional to $\chi_B(G^D(F_{sparse})) \cdot \sum_{i=1}^{p} \omega(F_i) +$ $\min(m,n) \cdot \omega(F_{p+1})$, which is much less than $\min(m,n) \cdot \omega(F) = \min(m,n) \cdot \sum_{i=1}^{p+1} \omega(F_i)$, because $\chi_B(G^D(F_{sparse})) \ll \min(m,n)$ in the presence of the sparsity of F^{sparse}.*

Example 4.1.2. *Define function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$:*

$$F = \underbrace{F_{sub} \circ F_{sub} \circ \cdots \circ F_{sub}}_{n\ F_{sub}\text{'s}}$$

*where the Jacobian matrix J^{sub} of F_{sub} is sparse, and nonzero only for entries J_{ij}^{sub} where $|i-j| \leqslant 1$. F's Jacobian matrix J is dense since $J = \prod_{i=1}^{n} J_i^{sub}$, hence sparsity techniques do not apply. With reference to **Example** 4.1.1, the overall computing time is proportional to $n \cdot \omega(F)$. Now suppose we applied the directed edge separator method on F first and generated n sub-functions $F_1, F_2, \cdots, F_n$, and further assume ideal separators are located, i.e., $F_1 = F_2 = \cdots = F_n = F_{sub}$, then $\chi_B(G^D(F_1)) = \chi_B(G^D(F_2)) = \cdots =$ $\chi_B(G^D(F_n)) = \chi_B(G^D(F_{sub}))) = 3$ [15]. Therefore the computational cost can be reduced to $\sum_{i=1}^{n} \chi_B(G^D(F_i)) \cdot \omega(F_i) = 3 \cdot \omega(F) \ll n \cdot \omega(F)$ for big n's.*

The function in **Example** 4.1.2 reflects the characteristics of many real world applications — their associated computational graphs are sparse, i.e., each variable takes at most two variables as input and on average serves as the input variable for few subsequent nodes. The whole function may appear to be 'dense', however by applying the directed edge separators method on it, the sparsity will show up gradually as number of separators increases. To see this we can consider an extreme case that number of separators is maximized — the function is broken down in to $\omega(F)$ sub-functions ($p+1 = \omega(F)$); each basic operation corresponds to a sub-function F_i, and hence corresponds to one row in the extended Jacobian matrix J_E. Because each basic operation has one output variable and at most two input variables defining the relationship between at most three variables, J_E is sparse with at most three nonzero entries in each row. In this case with 'finest granularity' partition, the bi-chromatic number is bounded by $\max_i (|\text{row}_i(J_E)|_{\text{NNZ}}) = 3$ [15].

As pointed out in **Section** 2.3, in practice we do not partition the computational graph into finest granularity due to its large space requirement. Instead we aim at sub-functions with reasonable size, balanced by the time requirement and the space requirement.

After suitable edge separators are located, we can take advantage of the hidden structure of functions. The acceleration is remarkable when applying sparsity techniques such as the bi-coloring method. In this chapter, **Section** 4.2 briefly reviews several sparsity techniques. **Section** 4.3 describes several typical problems with hidden structure. Details and specifications of the algorithms used in the numerical experiments are given in **Section** 4.4, and **Section** 4.5 contains the results of the numerical experiments.

4.2 A Brief Review of Sparsity Techniques

For a function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, if its Jacobian matrix $J \in \mathbb{R}^{m\times n}$ is sparse and the sparsity pattern is known, then sparsity techniques can be applied to improve the efficiency for calculating J. The improvement depends on the specific sparsity patterns, and in general is bigger for sparser J.

The key idea of all sparsity techniques is: instead of calculating J directly, calculate JV and/or W^TJ first for some 'thin' W and V, and then extract J from JV and/or W^TJ. (JV, W^TJ) is preferable because it can be obtained with less work compared with that needed to determine J directly.

Given an arbitary matrix $V \in n \times t_V$, the product JV can be directly computed using forward mode in time proportial to $t_V \cdot w(F)$; given an arbitary matrix $W \in m \times t_W$, the product W^TJ can be directly computed using reverse mode in time proportial to $t_W \cdot w(F)$[15, 25, 36]. If W and V are thin, the work required to obtain (JV, W^TJ) can be much less than that to obtain J directly, i.e., $(t_V + t_W) \cdot w(F) \ll \min(m, n) \cdot \omega(F)$.

The objective for sparsity techniques is: for given pattern of J, find (V, W) with small $t_V + t_W$, meanwhile maintain (JV, W^TJ)'s extractability for J. See the following examples for an illustration.

Example 4.2.1. *Suppose for a function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$, its Jacobian matrix J has the following pattern:*

$$J = \begin{bmatrix} J_{1,1} & 0 & 0 & \cdots & \cdots & 0 \\ J_{2,1} & J_{2,2} & 0 & & & \vdots \\ J_{3,1} & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ J_{n,1} & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix}.$$

Choose V to be the following $n \times t_V$ matrix where $t_V = 2$

$$V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{bmatrix}.$$

Then the product JV is

$$JV = \begin{bmatrix} J_{1,1} & 0 & 0 & \cdots & \cdots & 0 \\ J_{2,1} & J_{2,2} & 0 & & & \vdots \\ J_{3,1} & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ J_{n,1} & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} J_{1,1} & 0 \\ J_{2,1} & J_{2,2} \\ J_{3,1} & J_{3,3} \\ \vdots & \vdots \\ \vdots & \vdots \\ J_{n,1} & J_{n,n} \end{bmatrix}.$$

Because each nonzero entry of J is in JV as well, clearly J can be extracted from JV. Notice that the work required to calculate JV is $t_V \cdot \omega(F) = 2 \cdot \omega(F) \ll n \cdot \omega(F)$.

The method of solely using JV to retrieve J is known as *one-sided column method*, since JV is a compact way to store the nonzeros of J by compressing the columns. One-sided column method works very well for the above example, but it is not as useful for computing Jacobian matrices with dense rows, the following is an expamle:

Example 4.2.2. *Suppose for a function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$, its Jacobian matrix J has the following pattern:*

$$J = \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} & \cdots & \cdots & J_{1,n} \\ 0 & J_{2,2} & 0 & \cdots & \cdots & 0 \\ 0 & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & \cdots & 0 & J_{n,n}. \end{bmatrix}.$$

It is not very useful to use the one-sided column method here, because column contraction leads to summation of the nonzeros in the first row of J. These nonzero entries will not be retrievable after the summation. To avoid the summation, the only choice for V is $V = I$, the identity matrix, where $t_V = n$. There will be no gain on efficiency for such a V since $t_V \cdot \omega(F) = n \cdot \omega(F)$.

However if use $W^T J$ instead of JV to extract J, we will not have to sum on the nonzeros. Let W be the following $n \times t_W$ matrix where $t_W = 2$:

$$W = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{bmatrix}.$$

Then the product $W^T J$ is:

$$W^T J = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} & \cdots & \cdots & J_{1,n} \\ 0 & J_{2,2} & 0 & \cdots & \cdots & 0 \\ 0 & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix} = \begin{bmatrix} J_{1,1} & J_{1,2} & \cdots & J_{1,n} \\ 0 & J_{2,2} & \cdots & J_{n,n} \end{bmatrix}.$$

*Similar to **Example** 4.2.1, because each nonzero entry of J is in $W^T J$ as well, obviously J can be extracted from $W^T J$. Notice that the work required to calculate $W^T J$ is $t_W \cdot \omega(F) = 2 \cdot \omega(F) \ll n \cdot \omega(F)$.*

The method of solely using $W^T J$ to retrieve J is known as *one-sided row method*, since $W^T J$ is a compact way to store the nonzeros of J by compressing the rows.

It is easy to construct J with both dense rows and dense columns. For such patterns neither the one-sided column method nor the one-sided row method works well. However a combination of JV and $W^T J$ can usually bring much better performance. The following is an example.

Example 4.2.3. *Suppose for a function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$, its Jacobian matrix J has the following pattern:*

$$J = \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} & \cdots & \cdots & J_{1,n} \\ J_{2,1} & J_{2,2} & 0 & \cdots & \cdots & 0 \\ J_{3,1} & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ J_{n,1} & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix}.$$

The Jacobian matrix J is sparse. However the one-sided column method leads to the summation of the nonzeros in the first row, while the one-sided row method leads to the summation of the nonzeros in the first column. Neither of the two is helpful in this problem.

It turns out that if we use JV and W^TJ together, then the sparsity pattern can be facilitated to improve the efficiency again. Choose V and W to be:

$$V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{bmatrix}, W = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Then JV and W^TJ are the following respectively:

$$JV = \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} & \cdots & \cdots & J_{1,n} \\ J_{2,1} & J_{2,2} & 0 & \cdots & \cdots & 0 \\ J_{3,1} & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ J_{n,1} & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} J_{1,1} & \square \\ J_{2,1} & J_{2,2} \\ J_{3,1} & J_{3,3} \\ \vdots & \vdots \\ \vdots & \vdots \\ J_{n,1} & J_{n,n} \end{bmatrix},$$

$$W^TJ = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} & \cdots & \cdots & J_{1,n} \\ J_{2,1} & J_{2,2} & 0 & \cdots & \cdots & 0 \\ J_{3,1} & 0 & J_{3,3} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ J_{n,1} & 0 & \cdots & \cdots & 0 & J_{n,n} \end{bmatrix} = \begin{bmatrix} J_{1,1} & J_{1,2} & \cdots & J_{1,n} \end{bmatrix}.$$

The first column and the diagonal of J can be retrieved from JV, and the first row of J can be retrieved from W^TJ. Hence the full Jacobian matrix J can be extracted from the matrix pair (JV, W^TJ).

To obtain (JV, W^TJ), both forward mode and reverse mode are necessary. The overall computation time is proportional to $(t_V + t_W) \cdot \omega(F) = 3 \cdot \omega(F)$.

In general for an arbitary sparse function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, its pattern can be much more complicated than that in **Example** 4.2.3. We aim to find (V, W) such that $t_V + t_W$ is minimized, meanwhile J is retrievable from (JV, W^TJ).

The bi-coloring method is an algorithm that returns an almost optimal (V, W) pair for an arbitary pattern[15]. The performance of the bi-coloring method is proven by experiments, and it is being widely used in numerous applications.

4.3 Example Problems with Hidden Structure

Many functions have hidden structure. Here we describe two extreme examples: the dynamic system problem and the partial separability problem.

The dynamic system problem and the separability problem are represented using a set of sub-functions. This representation does not limit to these two problems; it is a general framework which can adapt various types of functions. This framework is as follows:

$$\left.\begin{array}{lcl}\text{solve for } y_1^{sub} & : & F_1^{sub}(x, y_1^{sub}) = 0 \\ \text{solve for } y_2^{sub} & : & F_2^{sub}(x, y_1^{sub}, y_2^{sub}) = 0 \\ \vdots & & \vdots \\ \text{solve for } y_p^{sub} & : & F_p^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) = 0 \\ \text{solve for } z & : & F_{p+1}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z = 0\end{array}\right\} \tag{4.1}$$

Notice that though it looks similar, the above system is different from that in (2.11). The above system is the definition of a function F, and is independent of the edge separators. In contrast, $F_1, F_2, \cdots, F_{k+1}$ in (2.11) is a decomposition of F based on the locations of the associated edge separators.

4.3.1 Dynamic System Problem

Definition 4.3.1. ***Dynamic System Problem (DS)*** *is a special case of* ***System*** *(4.1) where the sub-functions have the following form:*

$$\left.\begin{array}{lcl}\textit{solve for } y_1^{sub} & : & F_1^{sub}(x, y_1^{sub}) \equiv F_1^{sub}(x) - y_1^{sub} = 0 \\ \textit{solve for } y_2^{sub} & : & F_2^{sub}(x, y_1^{sub}, y_2^{sub}) \equiv F_2^{sub}(y_1^{sub}) - y_2^{sub} = 0 \\ \vdots & & \vdots \\ \textit{solve for } y_k^{sub} & : & F_p^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) \equiv F_{p-1}^{sub}(y_{p-1}^{sub}) - y_p^{sub} = 0 \\ \textit{solve for } z & : & F_{p+1}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z \equiv \bar{F}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z = 0\end{array}\right\} \tag{4.2}$$

Or equivalently,

$$\left.\begin{array}{lcl}\textit{solve for } y_i^{sub} & : & F_i^{sub}(y_{i-1}^{sub}) - y_i^{sub} = 0, \qquad i = 1, 2, \cdots, p \\ \textit{solve for } z & : & \bar{F}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z = 0\end{array}\right\} \tag{4.3}$$

where $y_0 = x$.

Suppose the directed edge separator method is applied to evaluate the extended Jacobian J_E, and sub-functions $F_1, F_2, \cdots, F_k, \bar{F}$ are located. If $\bar{F}$'s computational graph is a superset of $\bar{F}^{sub}$'s, i.e., $\vec{G}(\bar{F}) \supseteq \vec{G}(\bar{F}^{sub})$, then the Jacobian matrix of F described by equation (2.12) can be simplified to:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & -I & 0 & \cdots & \cdots & 0 \\ 0 & (F_2)_{y_1} & -I & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & \ddots & -I \\ (\bar{F})_x & (\bar{F})_{y_1} & \cdots & \cdots & \cdots & (\bar{F})_{y_k} \end{bmatrix}. \tag{4.4}$$

4.3.2 Partial Separability Problem

Definition 4.3.2. ***Partial Separability Problem (PS)*** *is a special case of equation (4.1) where the sub-functions have the following form:*

$$\left.\begin{array}{lcl} \textit{solve for } y_1^{sub} & : & F_1^{sub}(x, y_1^{sub}) \equiv F_1^{sub}(x) - y_1^{sub} = 0 \\ \textit{solve for } y_2^{sub} & : & F_2^{sub}(x, y_1^{sub}, y_2^{sub}) \equiv F_2^{sub}(x) - y_2^{sub} = 0 \\ \vdots & & \vdots \\ \textit{solve for } y_k^{sub} & : & F_p^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) \equiv F_{p-1}^{sub}(x) - y_p^{sub} = 0 \\ \textit{solve for } z & : & F_{p+1}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z \equiv \bar{F}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z = 0 \end{array}\right\} \tag{4.5}$$

Or equivalently,

$$\left.\begin{array}{lcl} \textit{solve for } y_i^{sub} & : & F_i^{sub}(x) - y_i^{sub} = 0, \qquad i = 1, 2, \cdots, p \\ \textit{solve for } z & : & \bar{F}^{sub}(x, y_1^{sub}, \cdots, y_p^{sub}) - z = 0 \end{array}\right\} \tag{4.6}$$

Suppose the directed edge separator method is applied to evaluate the extended Jacobian J_E, and sub-functions $F_1, F_2, \cdots, F_k, \bar{F}$ are located. If $\bar{F}$'s computational graph is a superset of $\bar{F}^{sub}$'s, i.e., $\vec{G}(\bar{F}) \supseteq \vec{G}(\bar{F}^{sub})$, the Jacobian matrix of F described by equation

(2.12) can be then simplified to:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & -I & 0 & \cdots & \cdots & 0 \\ (F_2)_x & 0 & -I & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ (F_k)_x & 0 & \cdots & \cdots & 0 & -I \\ (\bar{F})_x & (\bar{F})_{y_1} & \cdots & \cdots & \cdots & (\bar{F})_{y_k} \end{bmatrix}. \tag{4.7}$$

4.4 Pseudo-Code for Calculating the Extended Jacobian Matrix J_E

Below is a general framework, or pseudo-code, for calculating J_E.

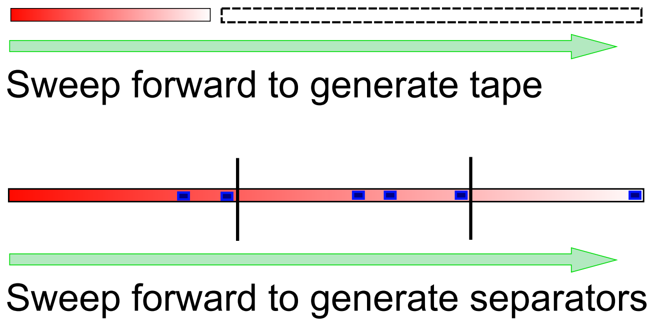
Algorithm 4.4.1. *The 'slice by slice' framework for calculating the extended Jacobian matrix J_E of function F is described as the following:*

```
1. Locate the directed edge separators and corresponding sub-functions $F_1, F_2, \cdots, F_k, F_{k+1}$ for $F$, $i \leftarrow 1$;
2. Apply forward sweep on $F_i$ to obtain $SP_i$, the pattern of $F_i$'s Jacobian matrix;
3. With sparsity pattern $SP_i$ obtained, apply sparsity techniques on $F_i$ to get the $i^{th}$ row-block $[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}, 0, \cdots, 0]$ of $J_E$. Save this row-block;
4. if $i < k+1$, then $i \leftarrow i+1$ and return to step 2, otherwise proceed to step 5;
5. Combine all the row-blocks to get the complete extended Jacobian matrix $J_E$.
```

4.4.1 Global Approach and Online Approach

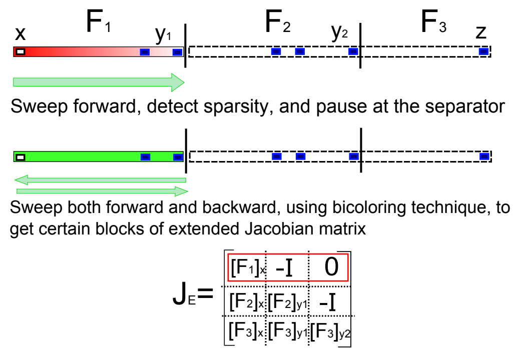
Before locating the directed edge separators, the computational graph/tape of function F must be generated first. There are two approaches to perform the generation tasks: the global approach and the online approach.

As illustrated in **Figure** 4.4.1, the global approach generates the complete tape first. Then the edge separators are located, and they are used to compute of J_E. Notice that the highlighted cells correpond to the intermediate variables[1] y_i's and output variable z defined in **Section** 2.2.



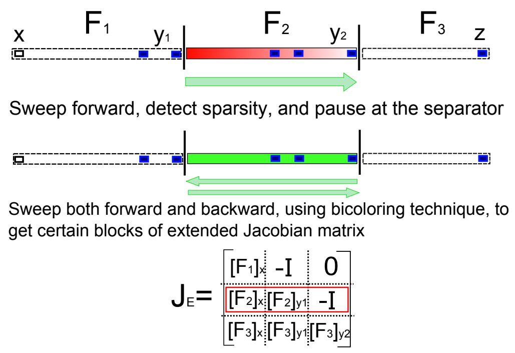
Sweep forward to generate tape
Sweep forward to generate separators

(a) Generation of the tape and the edge separators



x
F1
y1
F2
y2
F3
z
Sweep forward, detect sparsity, and pause at the separator
Sweep both forward and backward, using bicoloring technique, to get certain blocks of extended Jacobian matrix
JE=
[F1]x -I 0
[F2]x [F2]y1 -I
[F3]x [F3]y1 [F3]y2

(b) Calculation of the 1st row-block of J_E



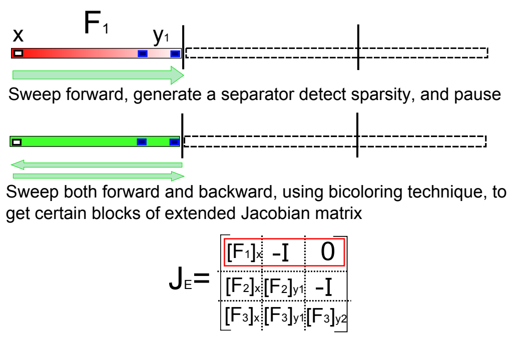
x
F1
y1
F2
y2
F3
z
Sweep forward, detect sparsity, and pause at the separator
Sweep both forward and backward, using bicoloring technique, to get certain blocks of extended Jacobian matrix
JE=
[F1]x -I 0
[F2]x [F2]y1 -I
[F3]x [F3]y1 [F3]y2

(c) Calculation of the 2nd row-block of J_E

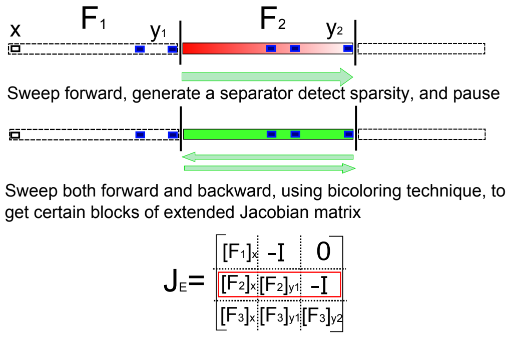
Figure 4.1: Global approach for generation of the tape

In contrast, the online approach generates the tape in an on-the-fly fashion, i.e., at any point of time only a partial-tape is stored. After a separator is located using this partial-tape, a row-block of J_E will be calculated using the separator information. Once the computation for the row-block is complete, this partial-tape is removed except the intermediate variables[1] (highlighted cells), and new partial-tapes will be generated. In the online approach the complete computational graph/tape is never stored. See **Figure** 4.4.1 for an illustration.

[1] The intermediate variables are those that will be needed again as inputs in later computations. Please refer to **Appendix** B.6 for the method to identify the intermediate variables.


x
F1
y1
Sweep forward, generate a separator detect sparsity, and pause
Sweep both forward and backward, using bicoloring technique, to get certain blocks of extended Jacobian matrix
JE=
[F1]x -I 0
[F2]x [F2]y1 -I
[F3]x [F3]y1 [F3]y2

(a) Generation of the 1^{st} partial-tape and the corresponding edge separator


x
F1
y1
F2
y2
Sweep forward, generate a separator detect sparsity, and pause
Sweep both forward and backward, using bicoloring technique, to get certain blocks of extended Jacobian matrix
JE=
[F1]x -I 0
[F2]x [F2]y1 -I
[F3]x [F3]y1 [F3]y2

(b) Generation of the 2^{nd} partial-tape and the corresponding edge separator

Figure 4.2: Online approach for generation of the tape

Since more information is accessible, the global approach can generate better separators compared to online approach. However the online approach is much more "space-friendly" than the global approach, since it stores at most one partial-tape each time. Global approach can be space costly, especially for computationally intense functions. However in some cases it may be worth the effort to use the global approach, because good separators can be used repeatedly. For example, given a function F whose Jacobian matrix $J(x)$ is desired at many different x's, it is better to use global approach to generate good separators first, and use them repeatedly for calculations at different x's.

4.4.2 Edge Separators Locating Methods in the Numerical Experiments

In **Chapter** 3 two methods were introduced: the weighted minimum edge separator method and the natural order edge separator method.The natural order separator method generates sub-functions corresponding to a consecutive sub-tape of the full tape. Furthermore, the reverse mode of ADMAT is tape-based, therefore it is more convenient to use the edge separators located by the natural order separator method.

Online version natural order edge separator is a combination of **Algorithm** 3.3.1 and online approach, it is stated as follows:

Algorithm 4.4.2. *Suppose a cost function f^{cost} is defined on a directed edge separator E_d, representing the quality of E_d in terms of separator size and partition ratio, and the*

function value decreases as the quality increases. Then the following are the steps finding the optimal natural order edge separators using online approach*:*

1. *Set partial-tape length parameter L, $i \leftarrow 1$;*

2. *Perform forward sweep on F, and generate the partial-tape T for F, pause when length of T reaches L;*

3. *Scan through tape T to find $j^* = \arg\min_j f^{cost}(E(\vec{G}(T(1:j), \vec{G}(T(j+1:end)))$;*

4. *$\vec{G}_i \leftarrow \vec{G}(T(1:j^*))$, compute and store the row-block $[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}, 0, \cdots, 0]$ where F_i is defined by $T(1:j^*)$;*

5. *Remove non-intermediate variable cells in $T(1:j^*)$ from T, resume generation of tape T, and append newly generated cells to the end of T until number of new cells reaches L or the computation of F is finished, $i \leftarrow i+1$;*

6. *Return to step 3 if computation of F is not completed, otherwise proceed to step 7;*

7. *Combine the row-blocks obtained to get J_E.*

Notice that in the above algorithm when locating the separators on the partial-tapes, we do not count the edges going from the already-generated partial-tapes to the partial-tapes to be generated. An example is the edge (c_i, c_e) going from an arbitrary cell c_i to the last cell c_e on the tape. When **Algorithm** 4.4.2 processes the partial-tape containing c_i, unless c_e is also in this partial-tape, the information of the edge (c_i, c_e) is not accessible since the last cell c_e is not yet generated. The miscounting of edge size does affect the quality of the separators located. The edge information is incomplete because we only store partial-tapes, and the drawback is unavoidable as long as we do not store the complete tape.

The above algorithm assumes most of the edges are short — for each edge its tail node and head node are close to each other on the tape. This is a reasonable assumption, because for most computations, most of the variables are used shortly after their generation. Under this assumption, the edge sizes counted by the above algorithm are good approximations to their real sizes, and **Algorithm** 4.4.2 will generate good edge separators.

If the space is the major concern, then subgraphs $\vec{G}_1, \vec{G}_2, \cdots, \vec{G_{k+1}}$ with equal size will help — see **Section** 2.3. We can modify **Algorithm** 4.4.2 slightly to generate equal-size separators:

Algorithm 4.4.3. *The following are the steps to find k evenly distributed edge separators for a given a function F:*

1. *Estimate the amount of computation for F, suppose the estimation is $\tilde{\omega}(F)$, set partial-tape length parameter L to be $\left\lceil \frac{\tilde{\omega}(F)}{k+1} \right\rceil$, $i \leftarrow 1$;*

2. *Perform forward sweep on F, and generate the partial-tape T for F, pause when length of T reaches L;*

3. *$\vec{G}_i \leftarrow \vec{G}(T)$, compute and store the row-block $[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}, 0, \cdots, 0]$ where F_i is defined by $\vec{G}_i$;*

4. *Remove non-intermediate variable cells from T, resume generation of tape, and append newly generated cells to the end of T until number of new cells reaches L or the computation of F is finished, $i \leftarrow i+1$;*

5. *Return to step 3 if computation of F is not completed, otherwise proceed to step 6;*

6. *Combine the row-blocks obtained to get J_E.*

Notice that L needs to be picked carefully as analyzed in **Section** 3.4. To see this we can consider the following two extreme cases: 1) if $L = 1$, then every cell is an intermediate cell and needs to be stored, and as a result the space requirement is not reduced since we need to store the complete graph; 2) if $L = |T|$, then the algorithm reduces to the traditional reverse mode, and the space requirement remains the same.

In the numerical experiments, **Algorithm** 4.4.3 is used to generate the tapes and locate directed edge separators.

Cost Analysis

In the above two algorithms, only a constant number of sweeps on the tape is needed to generate the edge separators. Therefore the running time for generating separator is proportional to $\omega(F)$, which is negligible compared to that needed to compute the extended Jacobian matrix. Therefore the running time for **Algorithm** 4.4.2 is $\omega(J_E) = O(\sum_{i=1}^{k+1}(\chi_B(G^D[F_i]) \cdot \omega(F_i)) + |J_E|_{\text{NNZ}}))$, and the running time for **Algorithm** 4.4.3 is $\omega(J_E) = O(L \cdot \sum_{i=1}^{k+1}(\chi_B(G^D[F_i])) + |J_E|_{\text{NNZ}}))$.

Since both algorithms store at most a length L partial-tape, the space required for these two algorithms is the same: $\sigma(J_E) = O(L + |J_E|_{\text{NNZ}})$.

4.4.3 Sparsity Techniques in Numerical Experiments

As reviewed in **Section** 4.2, there are several sparsity techniques available, including the one-sided row method, the one-sided column method, and the bi-coloring method. Among these three methods, the one-sided row method is appropriate, because an arbitary row-block $J_E^i = [(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}, 0, \cdots, 0] \in \mathbb{R}^{m_i \times n_i}$ has many more columns than rows, i.e., $m_i \ll n_i$. Hence the one-sided row method brings good performance since $t_W \cdot \omega(F_i) \leq m_i \cdot \omega(F_i) \ll n_i \cdot \omega(F_i)$[15, 25, 36].

The numerical experiments in **Section** 4.5 use the one-sided row method to calculate the extend Jacobian matrix J_E.

4.5 Computational Results

Space and time are the two major concerns in scientific computations.

According to **Section** 2.2, the space required for automatic differentiation is proportional to the maximum length of the tape. Therefore in each experiment, we will use the maximum length of tape as the indicator for space requirement.

Now let us analyze the computing time. First consider the directed edge separator method, if sub-functions $F_1, F_2, \cdots, F_{k+1}$ are located, and for each F_i, $W_i \in \mathbb{R}^{t_{W_i} \times p_i}$ is obtained by one-sided row method to help calculating the i^{th} row-block $[(F_i)_x, (F_i)_{y_1}, \cdots, (F_i)_{y_i}, 0, \cdots, 0]$ in J_E, then the work required to calculate this row-block is proportional to $t_{W_i} \cdot \omega(F_i)$. Therefore the work required for calculating all the row-blocks of J_E is proportional to $\sum_{i=1}^{k+1} t_{W_i} \cdot \omega(F_i)$. In contrast, the work required by the traditional reverse mode to calculate J is proportional to $m \cdot \omega(F) = \sum_{i=1}^{k+1} m \cdot \omega(F_i)$.

Furthermore, the equal-size partition (**Algorithm** 4.4.3) is used in numerical experiments, as a result, $\omega(F_1) = \omega(F_2) = \cdots \omega(F_{k+1})$. The work required by the directed edge separator method for getting J_E can be simplified to $\omega(F_1) \cdot \sum_{i=1}^{k+1} t_{W_i} = (k+1) \cdot \omega(F_1) \cdot \bar{t}_{W_i}$, where $\bar{t}_{W_i}$ is the average of t_{W_i} over $i = 1 : k+1$. Similarly the work required by the traditional reverse mode for getting J can be simplified to $\omega(F_1) \cdot \sum_{i=1}^{k+1} m = (k+1) \cdot \omega(F_1) \cdot m$.

$(\{t_{W_i}\}_{i=1:k+1}, m)$ and $(\bar{t}_{W_i}, m)$ are both running time indicators. When comparing running time of the two methods on a single problem, we will study the relationship between $\{t_{W_i}\}_{i=1:k+1}$ and m; when comparing their running time on a set of problems, we will study the relationship between $\bar{t}_{W_i}$ and m.

In this section we will provide the numerical results of above comparisons on the dynamic system problems and the partial separability problems.

4.5.1 The Dynamic System Problems

In the dynamic system problem (equation (4.3)) used in the experiments, the sub-functions F^i_{sub} and $\bar{F}_{sub}$ are defined by:

$$\begin{array}{ll} F^i_{sub}(y^{sub}_{i-1}) = y^{sub}_{i-1} + C^i \cdot y^{sub}_{i-1}, & i = 1, 2, \cdots, k \\ \bar{F}_{sub}(x, y^{sub}_1, \cdots, y^{sub}_k) = A(x)^{-1} \cdot y^{sub}_k, & \end{array} \tag{4.8}$$

where $y_0 = x$, C^i is a contant square matrix, $k = 50$, $A(x)$ is a nonsingular square matrix depending on input variables x. Notice that $x, y^{sub}_i, z \in \mathbb{R}^n$.

The DS problem can then be simplified to:

$$\left.\begin{array}{ll} y^{sub}_i = y^{sub}_{i-1} + C^i \cdot y^{sub}_{i-1}, & i = 1, 2, \cdots, 50 \\ z = A(x)^{-1} \cdot y^{sub}_{50} & \end{array}\right\} \tag{4.9}$$

By changing problem size n, we can get different instances of DS problem; by changing the partial-tape length parameter L, we can control the behaviour of directed edge separator method. The following are the results for $n = 64$ and $n = 144$ DS problems with $L = 100$ and $L = 200$:

		Traditional Reverse Mode	Directed Edge Separator method
$n = 64$	$L = 100$	2504	225
	$L = 200$		265
$n = 144$	$L = 100$	2504	225
	$L = 200$		265

Table 4.1: Maximum tape length – DS problems : $n = 64, 144$, $L = 100, 200$

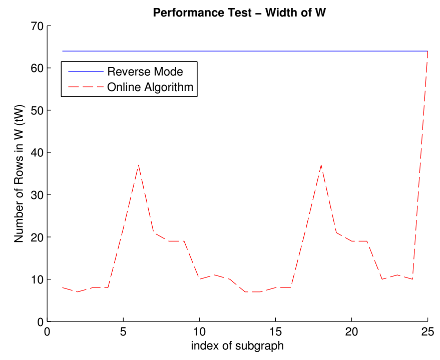
From **Table** 4.1 we can see that the space requirement is greatly reduced by the directed edge separator method.

The results on running time are shown in **Table** 4.2.

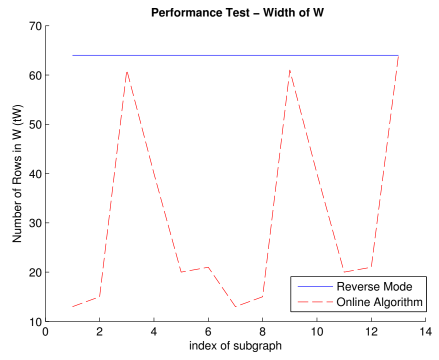
		m	$t_{W_i}, i = 1 : p + 1$
$n = 64$	$L = 100$	64	8 7 8 8 22 37 21 19 19 10 11 10 7 7 8 8 22 37 21 19 19 10 11 10 64
	$L = 200$		13 15 61 40 20 21 13 15 61 40 20 21 64
$n = 144$	$L = 100$	144	8 8 8 8 23 37 21 19 18 10 11 9 8 8 8 8 23 37 21 19 18 10 11 9 144
	$L = 200$		16 16 64 41 19 22 16 16 64 41 19 22 144

Table 4.2: Running time – DS problems : $n = 64, 144$, $L = 100, 200$

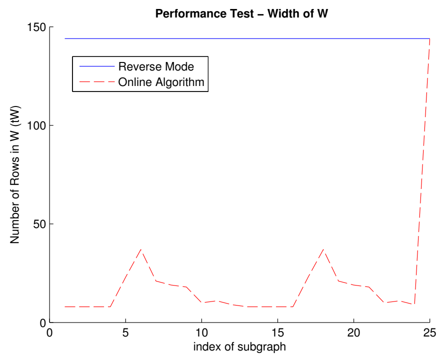
The above results on running time are also plotted in **Figure** 4.3. In the figures the x-axis is the indices of the sub-functions, and the y-value of the dash line is t_{W_i}, which is proportional to the work required to obtain the i^{th} row-block of J_E using the directed edge separator method. The solid line is at height m, which is proportional to the work required for the traditional reverse mode to process F_i.



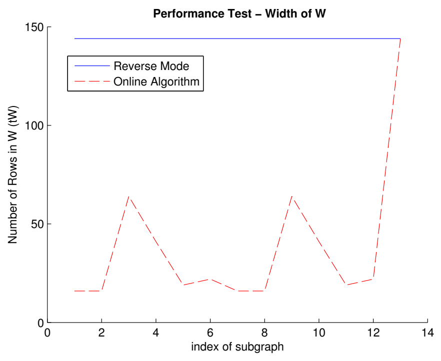
(a) $x, y_i^{sub}, z \in \mathbb{R}^{64}$ and $L = 100$



(b) $x, y_i^{sub}, z \in \mathbb{R}^{64}$ and $L = 200$



(c) $x, y_i^{sub}, z \in \mathbb{R}^{144}$ and $L = 100$



(d) $x, y_i^{sub}, z \in \mathbb{R}^{144}$ and $L = 200$

Figure 4.3: Running time for dynamic system problems

To see how the performance of the directed edge separator method changes as size of the problem varies, numerical experimental is done on a set of DS problems whose dimension n varies from 9 to 121. Maximum tape length is compared between the directed edge separator method and the traditional reverse mode. Besides, $\bar{t}_{W_i}$, the average value of t_{W_i}, is computed for each problem and compared with m.

We first choose $L = 100$, the test results on space and running time for these problems are listed in **Table** 4.3 and **Table** 4.4.

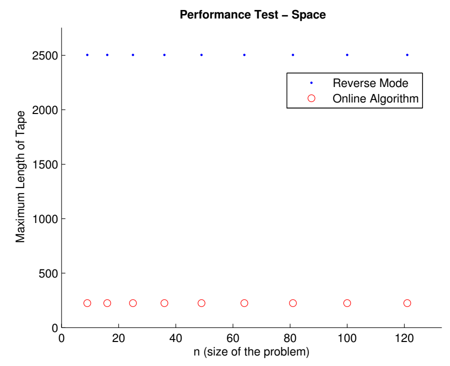
n	9	16	25	36	49	64	81	100	121
Traditional Reverse Mode	2054	2054	2054	2054	2054	2054	2054	2054	2054
Directed Edge Separator Method	225	225	225	225	225	225	225	225	225

Table 4.3: Maximum tape length – DS problems : $n = 9, \cdots, 121$, $L = 100$

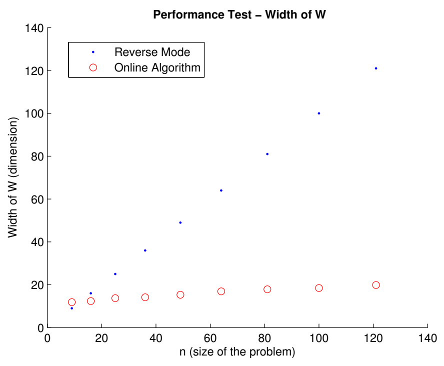
n	9	16	25	36	49	64	81	100	121
m	9	16	25	36	49	64	81	100	121
$\bar{t}_{W_i}$	11.88	12.40	13.76	14.16	15.32	16.92	17.88	18.48	19.88

Table 4.4: Running time – DS problems : $n = 9, \cdots, 121$, $L = 100$

The test results on space and running time are also plotted in **Figure** 4.4. From the graph we can see that the improvements are significant.



(a) Maximum tape length



(b) Running time

Figure 4.4: Space and running time for dynamic system problems with $L = 100$

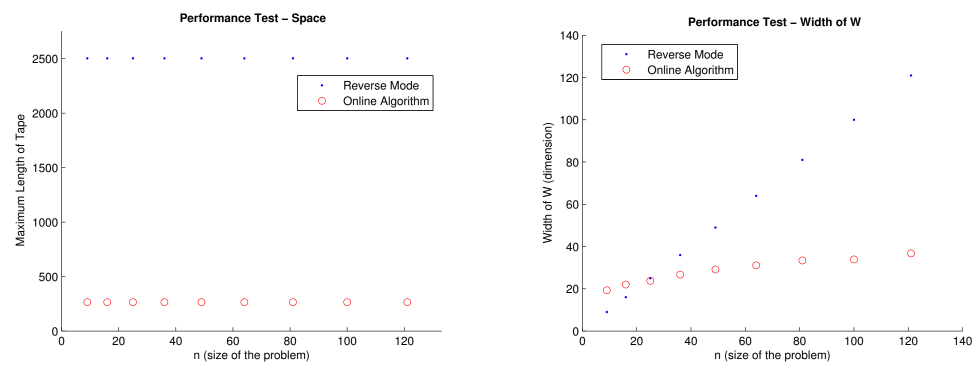
On the same set of problems, we let L to be 200, and repeat above experiments. The results are shown in **Table** 4.5 and **Table** 4.6.

n	9	16	25	36	49	64	81	100	121
Traditional Reverse Mode	2054	2054	2054	2054	2054	2054	2054	2054	2054
Directed Edge Separator Method	265	265	265	265	265	265	265	265	265

Table 4.5: Maximum tape length – DS problems : $n = 9, \cdots, 121,\ L = 200$

n	9	16	25	36	49	64	81	100	121
m	9	16	25	36	49	64	81	100	121
$\bar{t}_{W_i}$	19.31	22	23.77	26.69	29.15	31.08	33.38	33.92	36.77

Table 4.6: Running time – DS problems : $n = 9, \cdots, 121,\ L = 200$



(a) Maximum tape length (b) Running time

Figure 4.5: Space and running time for dynamic system problems with $L = 200$

The test results on space and running time are also plotted in **Figure** 4.5. Similar to the $L = 100$ case, the improvements are significant.

4.5.2 The Partial Separability Problems

Now consider the partial separability problem (equation (4.6)) in which F^i_{sub} and $\bar{F}_{sub}$ are defined by:

$$\begin{aligned} F^i_{sub}(x) &= x + C^i \cdot x, \qquad i = 1, 2, \cdots, k \\ \bar{F}_{sub}(x, y^{sub}_1, \cdots, y^{sub}_k) &= A(x)^{-1} \cdot \textstyle\sum_{i=1}^{50} y^{sub}_i, \end{aligned} \tag{4.10}$$

where C^i is a contant square matrix, $k = 50$, $A(x)$ is a nonsingular square matrix depending on input variables x. Notice that $x, y_i^{sub}, z \in \mathbb{R}^n$.

The PS problem can then be expressed as:

$$\left.\begin{array}{ll} y_i^{sub} = x + C^i \cdot x, & i = 1, 2, \cdots, 50 \\ z = A(x)^{-1} \cdot \sum_{i=1}^{50} y_i^{sub} & \end{array}\right\} \qquad (4.11)$$

By changing the problem size n, we can get different instances of PS problem; by changing the partial-tape length parameter L, we can control the behaviour of the directed edge separator method. The following are the results for $n = 64$ and $n = 121$ PS problems with $L = 100$ and $L = 200$:

		Traditional Reverse Mode	Directed Edge Separator method
$n = 64$	$L = 100$	3409	300
	$L = 200$		285
$n = 121$	$L = 100$	2504	300
	$L = 200$		285

Table 4.7: Maximum tape length – PS problems : $n = 64, 121,\ L = 100, 200$

From **Table** 4.7 we can see that the space requirement is greatly reduced by the directed edge separator method.

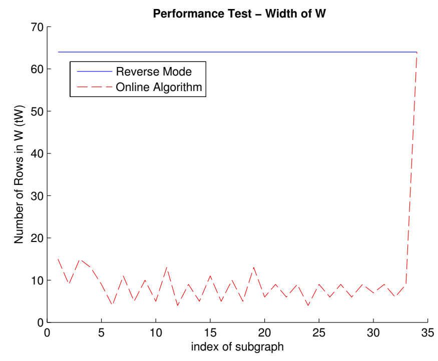
The results on running time are shown in **Table** 4.8.

		m	$t_{W_i}, i = 1 : p + 1$
$n = 64$	$L = 100$	64	15 9 15 13 9 4 11 5 10 5 13 4 9 5 11 5 10 5 13 6 9 6 9 4 9 6 9 6 9 7 9 6 9 64
	$L = 200$		9 13 4 5 5 4 5 5 5 6 6 4 6 6 7 6 64
$n = 144$	$L = 100$	144	14 8 11 14 8 5 10 4 11 4 14 5 8 5 10 5 10 5 15 6 9 6 9 5 7 6 7 6 8 7 8 7 8 121
	$L = 200$		9 15 5 4 4 5 5 5 5 6 6 5 6 6 7 7 121

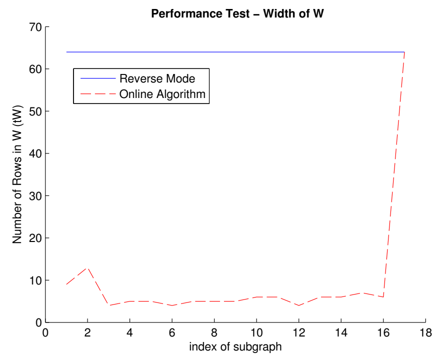
Table 4.8: Running time – PS problems : $n = 64, 121,\ L = 100, 200$

The above results are plotted in **Figure** 4.6. In this figure the x-axis is the index i for the sub-function F_i, and the y-value of the dash line is t_{W_i}, which is proportional to the work required to obtain the i^{th} row-block of J_E using the directed edge separator method.

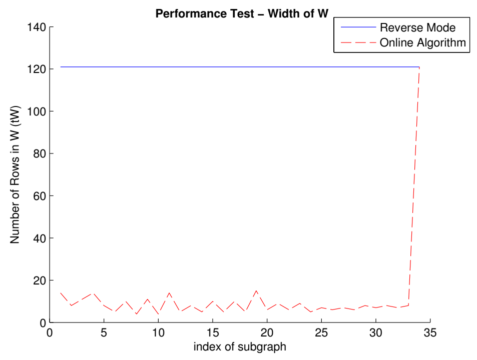
The solid line is at height m, which is proportional to the work required for the traditional reverse mode to process F_i.



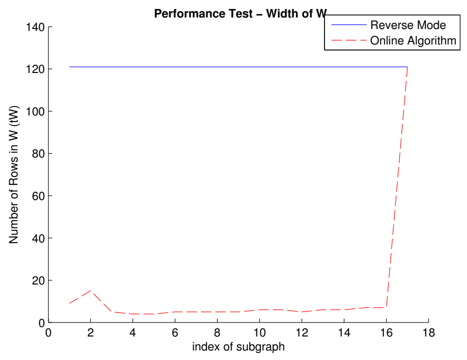
(a) $x, y_i^{sub}, z \in \mathbb{R}^{64}$ and $L = 100$



(b) $x, y_i^{sub}, z \in \mathbb{R}^{64}$ and $L = 200$



(c) $x, y_i^{sub}, z \in \mathbb{R}^{144}$ and $L = 100$



(d) $x, y_i^{sub}, z \in \mathbb{R}^{144}$ and $L = 200$

Figure 4.6: Running time for partial separability problems

To see how the performance of the directed edge separator method changes as size of the problem varies, numerical experimental is done on a set of PS problems whose dimension n varies from 9 to 121. Maximum tape length is compared between the directed edge separator method and the traditional reverse mode. Besides, $\bar{t}_{W_i}$, the average value of t_{W_i}, is computed for each problem and compared with m.

We first choose $L = 100$, the test results on space and running time for these problems are listed in **Table** 4.9 and **Table** 4.10.

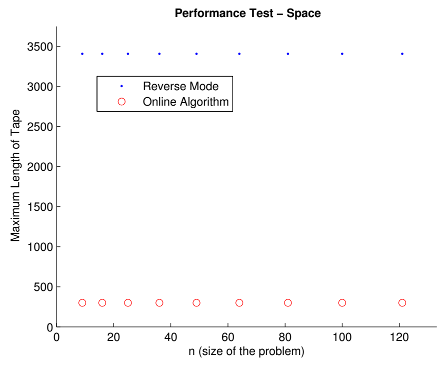
n	9	16	25	36	49	64	81	100	121
Traditional Reverse Mode	3409	3409	3409	3409	3409	3409	3409	3409	3409
Directed Edge Separator Method	300	300	300	300	300	300	300	300	300

Table 4.9: Maximum tape length – PS problems : $n = 9, \cdots, 121$, $L = 100$

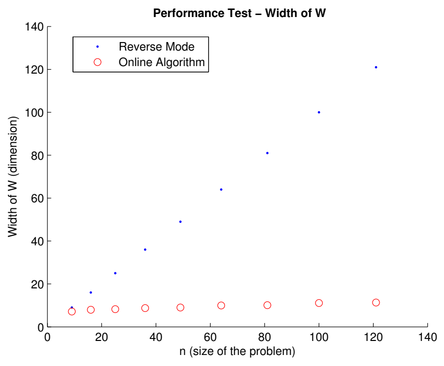
n	9	16	25	36	49	64	81	100	121
m	9	16	25	36	49	64	81	100	121
$\bar{t}_{W_i}$	7.15	7.97	8.24	8.74	9	9.97	10.09	11.12	11.35

Table 4.10: Running time – PS problems : $n = 9, \cdots, 121$, $L = 100$

The test results on space and running time are also plotted in **Figure** 4.7. From the graph we can see that the improvements are significant.



(a) Maximum tape length



(b) Running time

Figure 4.7: Space and running time for partial separability problems with $L = 100$

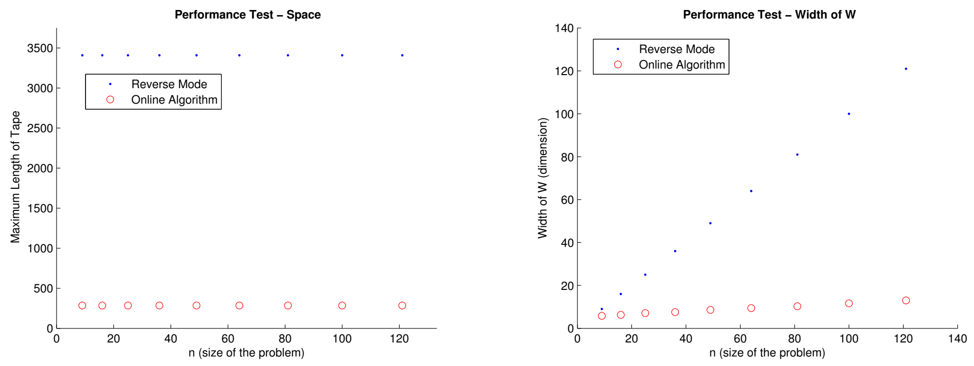
On the same set of problems, we let L to be 200, and repeat above experiments. The results are shown in **Table** 4.11 and **Table** 4.12.

n	9	16	25	36	49	64	81	100	121
Traditional Reverse Mode	3409	3409	3409	3409	3409	3409	3409	3409	3409
Directed Edge Separator Method	285	285	285	285	285	285	285	285	285

Table 4.11: Maximum tape length – PS problems : $n = 9, \cdots, 121$, $L = 200$

n	9	16	25	36	49	64	81	100	121
m	9	16	25	36	49	64	81	100	121
$\bar{t}_{W_i}$	5.82	6.29	7.12	7.59	8.59	9.41	10.29	11.65	13

Table 4.12: Running time – PS problems : $n = 9, \cdots, 121$, $L = 200$



(a) Maximum tape length (b) Running time

Figure 4.8: Space and running time for partial separability problems with $L = 200$

The test results on space and running time are also plotted in **Figure** 4.8. Similar to the $L = 100$ case, the improvements are significant.

4.6 Concluding Remarks

From the result, we can see that for both DS and PS problems, the directed edge separator method can greatly improve the efficiency, meanwhile reduce the space requirement. The improvements get even larger as the size of the problem increases. By reducing the partial-tape length constant L, computations may get more efficient depending on the specifics of the problems.

Chapter 5

The Special Case: the Structure of the Function is Provided

5.1 Introduction

In **Chapter** 4 we discussed how to use the the directed edge method to reveal the hidden structure for candidate function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, and hence improve the computation of $J(x)$ using the structure. Structure-revealing was necessary because we assumed they were not given explicitly. In contrast, this chapter considers a different case that the structure of the function is provided.

One trivial approach is: instead of trying to find the structure, take the provided one as an input for the directed edge method, and use the edge separator method to calculate J_E. It is reasonable to assume higher quality of the provided structures, hence this approach usually leads to better results. Since this trivial approach is essentially the edge separator method, which was investigated thoroughly in previous chapters, we will not repeat the analysis here.

This chapter mainly discusses a different method, called *S-2 algorithm*, which facilitate the structures in a 'reverse' fashion. The S-2 algorithm was proposed by Thomas Coleman and Wei Xu[20]. For an arbitary function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, it performs extremely well if m is small, i.e., $m \ll \bar{t}_{W_i}$. The following are the basic ideas.

Suppose for a function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, its structure is given by:

$$\left.\begin{array}{lcl}
\text{solve for } y_1 & : & F_1(x) - y_1 = 0 \\
\text{solve for } y_2 & : & F_2(x, y_1) - y_2 = 0 \\
\vdots & & \vdots \\
\text{solve for } y_k & : & F_p(x, y_1, \cdots, y_{p-1}) - y_p = 0 \\
\text{solve for } z & : & \bar{F}(x, y_1, \cdots, y_p) - z = 0
\end{array}\right\} \tag{5.1}$$

As shown in **Section 2.3**, the extended Jacobian matrix J_E of F is

$$J_E \triangleq \left[\begin{array}{c|cccccc}
J_x^1 & -I & 0 & \cdots & \cdots & 0 \\
J_x^2 & J_{y_1}^2 & -I & \ddots & & \vdots \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & & \ddots & \ddots & 0 \\
J_x^p & J_{y_1}^p & \cdots & \cdots & J_{y_{p-1}}^p & -I \\
\hline
\bar{J}_x & \bar{J}_{y_1} & \bar{J}_{y_2} & \cdots & \cdots & \bar{J}_{y_p}
\end{array}\right] = \left[\begin{array}{cc} A & B \\ C & D \end{array}\right],$$

and $J = C - DB^{-1}A$.

Since using J_E we can always calculate for J, obviously J_E contains at least as much information as J does; in fact J_E contains more information needed to obtain J. This property of J_E is the essential to this method: instead of calculating the complete J_E, we calculate for a 'contracted' version of J_E, which can be much cheaper to compute, and then use this 'contracted' J_E to obtain J. The formal description is as follows.

Define $W = [W_1, W_2, \cdots, W_p] = D \cdot B^{-1}$. Obviously $D = W \cdot B$, and hence,

$$D = [\bar{J}_{y_1}, \bar{J}_{y_2}, \cdots, \bar{J}_{y_p}] = [W_1, W_2, \cdots, W_p] \cdot \begin{bmatrix}
-I & 0 & \cdots & \cdots & 0 \\
J_{y_1}^2 & -I & \ddots & & \vdots \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
\vdots & & \ddots & \ddots & 0 \\
J_{y_1}^p & \cdots & \cdots & J_{y_{p-1}}^p & -I
\end{bmatrix}. \tag{5.2}$$

By taking the transpose of (5.2),

$$\begin{bmatrix} -I & (J^2_{y_1})^T & \cdots & \cdots & (J^p_{y_1})^T \\ 0 & -I & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & (J^p_{y_{p-1}})^T \\ 0 & \cdots & \cdots & 0 & -I \end{bmatrix} \begin{bmatrix} W_1^T \\ W_2^T \\ \vdots \\ \vdots \\ W_p^T \end{bmatrix} = \begin{bmatrix} \bar{J}^T_{y_1} \\ \bar{J}^T_{y_2} \\ \vdots \\ \vdots \\ \bar{J}^T_{y_p} \end{bmatrix}, \tag{5.3}$$

hence

$$J = C - DB^{-1}A = C - WA = \bar{J}_x - (W_1 J^1_x + W_2 J^2_x + \cdots + W_{p-1} J^{p-1}_x + W_p J^p_x). \tag{5.4}$$

Given (5.3) and (5.4), it is now possible to work backward through (5.4) to compute J as the following.

1. From the last row-block of (5.3) it is clear that $W_p = -\bar{J}_{y_p}$;
2. Apply reverse mode of automatic differentiation on $F_p(x, y_1, y_2, \cdots, y_{p-1})$ with matrix W_p^T to obtain $W_p J^p_x$ and $[W_p J^p_{y_1}, \cdots, W_p J^p_{y_{p-1}}]$;
3. Use the second last row-block of (5.3) to solve for $W_{p-1} = -\bar{J}_{p_1} + W_p J^p_{y_{p-1}}$;
4. Apply reverse mode of automatic differentiation on $F_{p-1}(x, y_1, y_2, \cdots, y_{p-2})$ with matrix $W^T_{p_1}$ to obtain $W_{p-1} J^{p-1}_x$ and $[W_{p-1} J^{p-1}_{y_1}, \cdots, W_{p-1} J^{p-1}_{y_{p-2}}]$;
5. Use the third last row-block of (5.3) to solve for $W_{p-2} = -\bar{J}_{p_2} + W_{p-1} J^{p-1}_{y_{p-2}} + W_p J^p_{y_{p-2}}$;
6. Repeat above steps until $W_1 J^1_x$ is obtained;
7. Use (5.4) to calculate J.

A formal description of the above idea as follows.

Algorithm 5.1.1. *Structured Jacobian calculation (S-2 Algorithm) can described as the following steps.*

1. *Use (5.1) to evaluate the value of intermediate variables* $y_1, \cdots, y_p$*;*
2. *Apply reverse mode of AD on* $z = \bar{F}(x, y_1, \cdots, y_p)$ *to obtain* $[\bar{J}_x, \bar{J}_{y_1}, \cdots, \bar{J}_{y_p}]$*;*

```
3. Compute the Jacobian matrix $J$ from $J_E$ using (5.1).
   (a) $J \leftarrow \bar{J}_x$, $T_i \leftarrow 0$ for $i = 1 : p$;
   (b) For $j = p, p-1, \cdots, 1$:
       i. $W_j \leftarrow -\bar{J}_{y_j} + T_j$;
       ii. Apply reverse mode of AD on $y_j = F_j(x, y_1, \cdots, y_{j-1})$ with matrix $W_j^T$ to
           obtain $W_j \cdot [J_x^j, J_{y_1}^j, \cdots, J_{y_{j-1}}^j]$;
       iii. $T_i \leftarrow T_i + W_j J_{y_i}^j$ for $i = 1, \cdots, j-1$;
       iv. $J \leftarrow J - W_j J_x^j$.
```

5.1.1 Cost Analysis

In S-2 Algorithm (**Algorithm** 5.1.1) the major computation is for the reverse mode of AD applied on $\bar{F}, F_p, \cdots, F_1$. For $\bar{F}$ the work required is proportional to $m \cdot \omega(\bar{F})$; for an arbitary F_i, we notice that the reverse mode is applied with matrix W_i^T, which has m columns, hence the work required is proportional to $m \cdot \omega(F_i)$. Therefore the overall work required for computing J is proportional to $m \cdot (\omega(\bar{F}) + \sum_{i=1}^{p} \omega(F_i)) = m \cdot \omega(F)$. It is clear that this algorithm works well for small m, and works best for $m = 1$.

The S-2 Algorithm has the same running time as that of the traditional reverse mode, however we notice that this algorithm makes huge improvement on space requirements. In S-2 Algorithm, the computation of $\bar{F}, F_p, \cdots, F_1$ are performed separately — at most one sub-function has its tape stored in memory at any momment. As a result the space requirement is proportional to the peak usage, which is $\max(\{\omega(\bar{F})\} \cup \{\omega(F_i)\}_{i=1:p})$. Compared to $\omega(F)$, the space requirement of the reverse mode, spaced required by S-2 Algorithm can be much smaller. To see this, consider the example that $\bar{F}$ and F_i's require a same amount of work to evaluate, i.e., $\omega(\bar{F}) = \omega(F_1) = \cdots = \omega(F_p) = \frac{\omega(F)}{p+1}$. Obviously $\max(\{\omega(\bar{F})\} \cup \{\omega(F_i)\}_{i=1:p}) = \frac{\omega(F)}{p+1}$.

5.2 Computational Results

This section compares the performance between the reverse mode and S-2 Algorithm. More specifically, space and running time experiments are performed on the DS problems and the GSP problems.

5.2.1 The Dynamic System Problem

In the dynamic system problem (equation (4.3)) used in the experiments, the sub-functions F^i_{sub} and $\bar{F}_{sub}$ are defined by:

$$\begin{aligned} &F^i_{sub}(y^{sub}_{i-1}) = y^{sub}_{i-1} + C^i \cdot y^{sub}_{i-1}, \qquad i = 1, 2, \cdots, k \\ &\bar{F}_{sub}(x, y^{sub}_1, \cdots, y^{sub}_k) = \left\| x + \sum_{i=1}^k y^{sub}_i \right\|^2_2 \end{aligned} \tag{5.5}$$

where $y_0 = x$ and C^i are constant square matrices. Notice that $x, y_i, z \in \mathbb{R}^n$.

The DS problem can then be simplified to:

$$\left.\begin{aligned} &y^{sub}_i = y^{sub}_{i-1} + C^i \cdot y^{sub}_{i-1}, \qquad i = 1, 2, \cdots, k \\ &z = \left\| x + \sum_{i=1}^k y^{sub}_i \right\|^2_2 \end{aligned}\right\} \tag{5.6}$$

where n and k are adjustable. By changing (n, k), we can get different instances of DS problem. **Table** 5.1 and 5.2 shows the results on space and time respectively for the $n = 20, k = 50, \cdots, 1000$ instances.

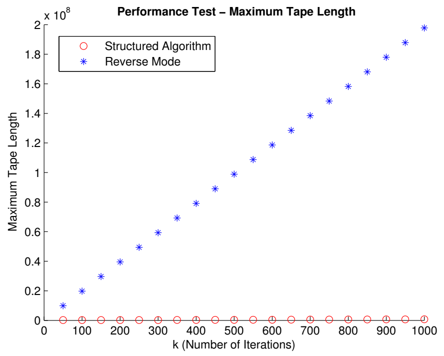
k	50	100	150	200	250	300	350	400	450	500
The reverse mode	9.889	19.78	29.66	39.55	49.44	59.33	69.22	79.11	88.99	98.88
Algorithm 5.1.1	0.198	0.199	0.200	0.201	0.202	0.229	0.267	0.305	0.343	0.381
k	550	600	650	700	750	800	850	900	950	1000
The reverse mode	108.8	118.7	128.5	138.4	148.3	158.2	168.1	178.0	187.9	197.8
Algorithm 5.1.1	0.419	0.457	0.495	0.533	0.571	0.609	0.647	0.685	0.723	0.761

Table 5.1: Maximum tape length ($\times 10^6$) – DS problems : $n = 20,\ k = 50, \cdots, 1000$

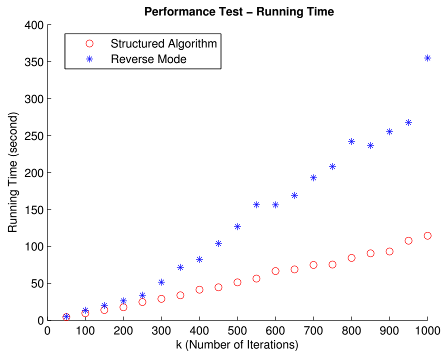
k	50	100	150	200	250	300	350	400	450	500
The reverse mode	5.23	13.63	20.08	26.36	34.07	51.65	71.73	82.53	103.93	126.80
Algorithm 5.1.1	4.42	9.65	14.05	17.77	24.81	29.14	33.91	41.53	44.75	51.37
k	550	600	650	700	750	800	850	900	950	1000
The reverse mode	156.4	156.3	169.0	192.9	208.0	242.0	236.5	255.1	267.7	354.9
Algorithm 5.1.1	56.58	66.72	68.97	75.02	75.53	84.44	90.74	93.21	107.8	114.5

Table 5.2: Running time (second) – DS problems : $n = 20,\ k = 50, \cdots, 1000$

The results are also plotted in **Figure** 5.1. From the figures we can see that the improvements of the S-2 Algorithm are significant.



(a) Maximum tape length



(b) Running time

Figure 5.1: Performace comparison on the DS problems with $n = 20$, $k = 50, \cdots, 1000$

5.2.2 The Partial Separability Problem

In the partial separability problem (equation (4.6)) used in the experiments, the subfunctions F^i_{sub} and $\bar{F}_{sub}$ are defined by:

$$\begin{aligned} F^i_{sub}(y^{sub}_{i-1}) &= x + C^i \cdot y^{sub}_{i-1}, \qquad i = 1, 2, \cdots, k \\ \bar{F}_{sub}(x, y^{sub}_1, \cdots, y^{sub}_k) &= \left\| x + \textstyle\sum_{i=1}^k y^{sub}_i \right\|_2^2 \end{aligned} \tag{5.7}$$

where C^i's are constant square matrices. Notice that $x, y_i, z \in \mathbb{R}^n$.

The PS problem can then be simplified to:

$$\left.\begin{aligned} y^{sub}_i &= x + C^i \cdot y^{sub}_{i-1}, \quad i = 1, 2, \cdots, k \\ z &= \left\| x + \textstyle\sum_{i=1}^k y^{sub}_i \right\|_2^2 \end{aligned}\right\} \tag{5.8}$$

where n and k are adjustable. By changing (n, k), we can get different instances of the PS problems. **Table** 5.3 and 5.4 shows the results on space and time respectively for the $n = 20, k = 50, \cdots, 1000$ instances.

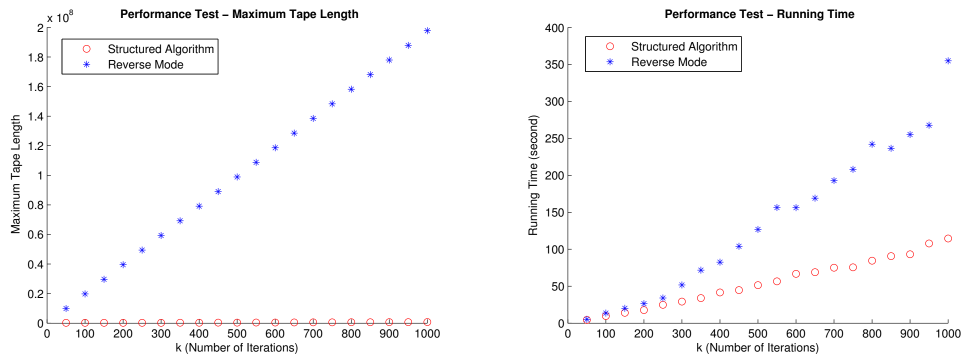
k	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800	850	900	950	1000
The reverse mode	9.889	19.78	29.66	39.55	49.44	59.33	69.22	79.11	88.99	98.88	108.8	118.7	128.5	138.4	148.3	158.2	168.1	178.0	187.9	197.8
Algorithm 5.1.1	0.198	0.199	0.200	0.201	0.202	0.229	0.267	0.305	0.343	0.381	0.419	0.457	0.495	0.533	0.571	0.609	0.647	0.685	0.723	0.761

Table 5.3: Maximum tape length $(\times 10^6)$ – PS problems : $n = 20$, $k = 50, \cdots, 1000$

k	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800	850	900	950	1000
The reverse mode	4.485	11.13	18.16	25.22	35.06	52.71	66.17	88.68	108.1	129.2	137.4	168.3	173.7	222.3	227.3	220.0	259.0	272.4	338.7	310.3
Algorithm 5.1.1	5.174	9.474	15.38	18.58	23.94	28.13	32.05	39.60	43.75	49.03	53.04	64.82	66.76	73.63	75.43	86.01	90.87	95.80	105.7	114.5

Table 5.4: Running time (second) – PS problems : $n = 20$, $k = 50, \cdots, 1000$

The test results on space and running time are also plotted in **Figure** 5.2. From the graph we can see that the improvements are significant.



(a) Maximum tape length (b) Running time

Figure 5.2: Performace comparison on the DS problems with $n = 20$, $k = 50, \cdots, 1000$

5.3 Concluding Remarks

By examination of our computational results, we see that the improvement regarding the space requirement is huge. Although theory predicts the work required for the reverse mode and the S-2 algorithm is the same, numerical experiments show that the S-2 Algorithm performs much better. One reason is the S-2 Algorithm has higher memory locality — computations are done within a smaller piece of memory, which saves the time for searching and accessing the data.

A drawback of S-2 Algorithm is it needs the user to manually identify the structures, i.e., provide the computer code of the sub-functions $F_1, \cdots, F_p, \bar{F}$. Hence it is not as "user-friendly" as the traditional reverse mode or the directed edge separator method.

Chapter 6

Conclusions and Future Work

Our experiments and analysis indicate that separation of nonlinear systems with use of directed edge separators can significantly reduce space requirement and computing time. Continued research and development along these lines is recommended:

- Fat computational graphs pose challenges to our current separator locating schemes. More research is needed here.
- We have not focused on the expense of locating the edge separators. Note that this operation need only to be computed once for a nonlinear system, this cost is amortized over many iterations. Nevertheless, research on efficiency of locating edge separators is required.
- The amortization remarks above assume that the structure of F is invariant with x. This is not always the case. Consider the following simple example:

$$f(x) = \begin{cases} 0 & x < 0 \\ f_0(x) & x \geq 0 \end{cases}$$

  where f_0 is an extremely complicated function. Then one can expect computational graphs of $f(-1)$ and $f(1)$ are totally different. Research is required regarding structures that vary with x.
- The implementation of a directed-edge-separator-method software package is recommended.

APPENDICES

Appendix A

Sparsest Cut

For an undirected graph G, its sparsest cut $E_s(G_1, G_2)$ is define to be

$$E_s(G_1, G_2) = \arg \min_{E(G_1,G_2)} \frac{|E(G_1, G_2)|}{\min\left(|G_1|, |G_2|\right)}$$

where $|E(G_1, G_2)|$ is number of edges crossing two subgraphs G_1 and G_2, $|G|$ denotes number of nodes in graph G [1].

Definition of sparsest cut fits our requirements for cutsets very well. This problem was well studied and there are already several approximate algorithm, which used linear programming/semidefinite programming techniques. However among these known algorithms, the best one is of $O(n^2)$ running time [1], which is still too slow in this application to be useful in practice (recall: $n \sim$ number of fundamental operations). As mentioned before, sometimes we run out of fast memory while using automatic differentiation. This means the computational graph's size is of the scale of a PC's memory. A quadratic algorithm is absolutely not acceptable even if constant is small. So we then turn to seek other approximation algorithms, with linear or near linear running time.

Appendix B

Generation of Computational Graphs

B.1 A computational graph view of AD

Automatic Differentiation (AD) is a joint field of computer science and applied mathematics, focusing on numerical differentiation of nonlinear functions. Its remarkable advantages on speed and accuracy, especially for certain types of functions, make AD widely used in financial applications, optimization algorithms, and other scientific computing problems. AD has experienced fast growth in the recent 20 years, during which many improvements and new techniques have been developed, including sparsity techniques[15, 25, 36], structure detection, etc.

Automatic differentiation can differentiate a function given its source code. There are two main modes of AD, forward mode and reverse mode. When forward mode is applied, the derivative and the value of the function are calculated simultaneously. When the evaluation of function reaches its end, the derivative is automatically obtained, (which is the reason this method is called automatic differentiation).

AD is different from finite difference, the traditional numerical differentiation, because AD essentially uses chain rule instead of numerical estimation. AD is also different from symbolic differentiation since AD computes numerically not algebraically.

Compared to symbolic computation, AD is much more practical, especially when the candidate function is provided in the form of computer code. In this case it is almost impossible to retrieve the expression of the derivative. While compared to finite difference, AD is much more efficient when the candidate function is sparse, or has high-dimension input and low-dimension output. Moreover AD is more accurate: it does not introduce any truncation error. To illustrate this we can consider a very common example, the derivative-based algorithm for single objective value optimization, in which all above conditions are true. In this problem AD will notably outperforms the other two when calculating gradients.

B.1.1 Basic Ideas

The evaluation of any function can be broken down into a series of basic operations. AD differentiates each basic operation symbolically, calculates the actual value of derivatives using the expression obtained, put them together using chain rule, and then obtains the final derivative of output variables with respect to input variables. Consider the following simple example:

Example B.1.1. *AD computation for $f : \mathbb{R} \mapsto \mathbb{R}$*

$$z = f(x) = x + \sin x \tag{B.1}$$

If break it down into basic operations, this system is equivalent to:

$$\begin{cases} y & = & \sin x \\ z & = & x + y \end{cases}$$

It can also be represented using a graph, where each node corresponds to a variable, and edges correspond to basic operations:

The differentiation of basic operations is easy. In this case they are the following:

$$\begin{cases} \frac{\partial y}{\partial x} & = & \cos x \\ \frac{\partial z}{\partial x} & = & \frac{\partial z}{\partial y} = 1 \end{cases}$$

Since there are only very limited types of basic operations, their symbolic expressions are usually stored in AD packages. They are ready to be plugged in with actual numbers. Suppose we want to obtain the gradient at $x = 2$, automatic differentiation will keep an addition attribute *deriv*, which store the derivative of current variable with respect to input variable x ($\frac{\mathrm{d}y_i}{\mathrm{d}x}$), on corresponding node, together with the value. At the end of the calculation the value of *deriv* corresponding to output variable z would be the overall objective derivative $\frac{\mathrm{d}z}{\mathrm{d}x}$ since the current variable $y_i = z$. The steps of forward mode is illustrated by **Figure.** B.1.1.

Reverse mode can reach the same result as well. It also keeps an extra attribute *deriv* in each node to help calculate the derivative. However reverse mode differs from forward

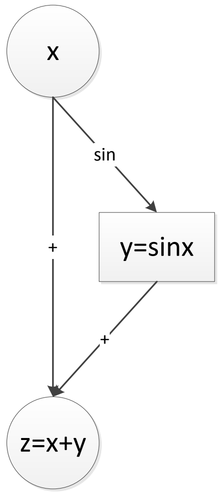


Figure B.1: The computational graph of **Example.** B.1.1

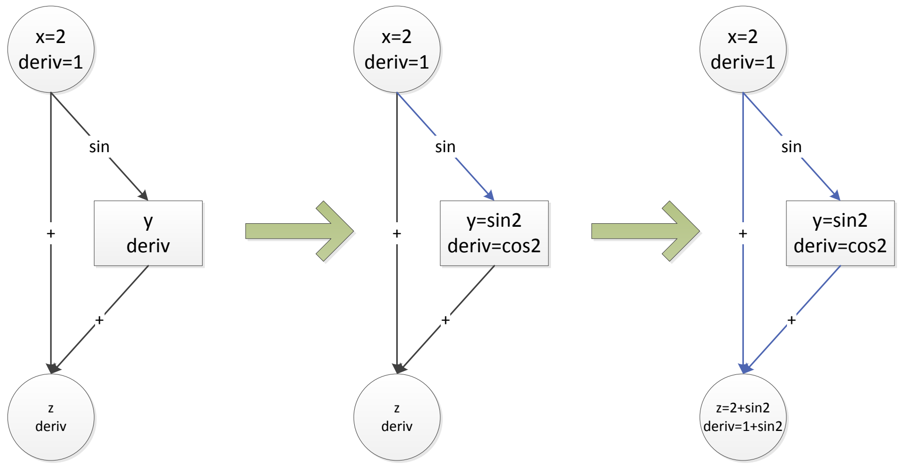
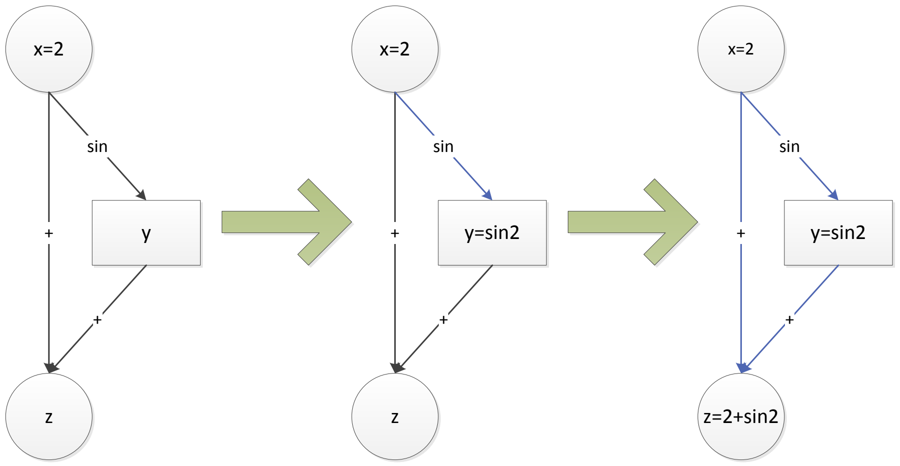
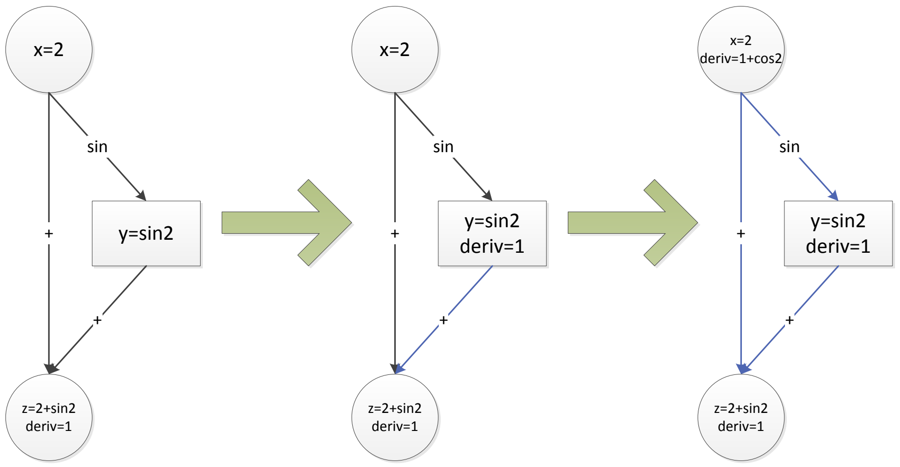


Figure B.2: Forward mode computation of **Example.** B.1.1



(a) Phase 1: calculation of values



(b) Phase 2: calculation of *deriv*

Figure B.3: Reverse mode computation of **Example.** B.1.1

mode in the sense that *deriv* stores $\frac{\mathrm{d}z}{\mathrm{d}y_i}$, the derivate of output variable $z$with respect to the current variable y_i. Another difference is the calculation of *deriv* is performed after the calculation of values. On the contrary in forward mode these calculations are performed simultaneously:

To run reverse mode stage 2, the record of calculation in stage 1 is mandatory. In this example the record is the computational graph generated at the end of stage1. In practice instead of the visual computational graph, a list of historical values and operations (which is equivalent to the graph) is generated as the record. Conventionally we call this list the *tape* of the computation. In stage 1, the computation starts from the beginning of the tape, and stops at the end of the tape, hence we say there is one *forward sweep*. In stage 2 the calculation starts in the other direction from the end of the tape, and stops at the beginning. Similarly we say there is one *reserve sweep* involved.

Obviously no reverse sweep is needed in forward mode. As a result it is not necessary to generate the tape in forward mode, which is only used in reverse sweep. This implies forward mode saves space compared to reverse mode. The space reduction can be considerably large especially when calculating for complicated functions, because the size of the tape is proportional to the number of operations performed. However reverse mode over performs forward mode in speed in certain cases, in particular, for candidate functions $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ where $n > m$.

B.1.2 Computational Cost Analysis

To see the computational cost of forward mode and reverse mode for functions $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ where $n > m$, first consider the following simple example with $n = 2$ and $m = 1$:

Example B.1.2. *AD computation for $F : \mathbb{R}^2 \mapsto \mathbb{R}$*

$$z = F(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = x_1 \cdot x_2 + \sin x_1 \tag{B.2}$$

If break it down into basic operations, this system is equivalent to: The differentiation of basic operations is easy. In this case they are the following:

$$\begin{cases} y_1 & = & x_1 \cdot x_2 \\ y_2 & = & \sin x_1 \\ z & = & y_1 + y_2 \end{cases}$$

It can also be represented using a graph as shown in **Figure.** B.4.

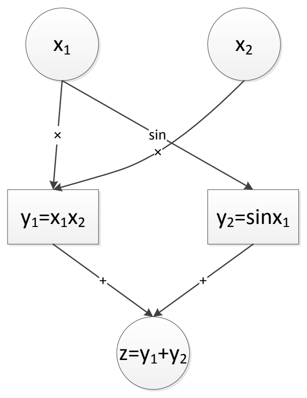


Figure B.4: The computational graph of **Example.** B.1.2

Forward Mode

Suppose we want to obtain the gradient at $x_1 = 2, x_2 = 3$, if using forward mode, then to get the gradient $\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \end{bmatrix}$ we need to apply forward sweep twice. By setting *deriv* at x_1 to be $\frac{\partial x_1}{\partial x_1} = 1$ and at x_2 to be $\frac{\partial x_2}{\partial x_1} = 0$, in later computation *deriv* of an arbitrary variable y_i will be updated by chain rule so that its value is $\frac{\partial y_i}{\partial x_1}$. At the end of computation $y_i = z$ and we can extract *deriv* from node to get $\frac{\partial z}{\partial x_1}$ as shown in **Figure.** B.5.

Similarly a second forward sweep can be applied to obtain $\frac{\partial z}{\partial x_2}$. Before the computation we need to set *deriv* at x_1 to be $\frac{\partial x_1}{\partial x_2} = 0$ and *deriv* at x_2 to be $\frac{\partial x_2}{\partial x_2} = 1$ as shown in **Figure.** B.6.

Combine the results from the two forward sweeps we can get the gradient:

$$\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 3 + \cos 2 \\ 2 \end{bmatrix}$$

Reverse Mode

The reverse mode needs a forward sweep first to construct the tape as shown in **Figure.** B.7.

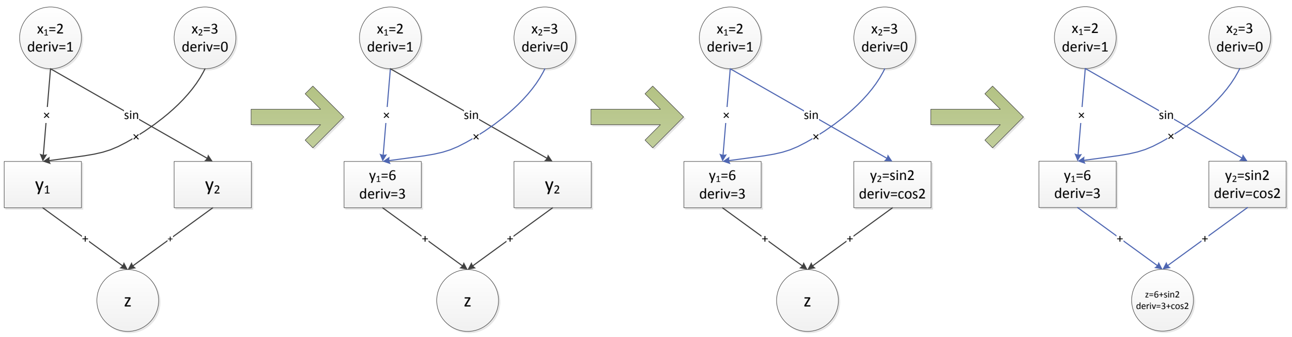


Figure B.5: Forward mode of **Example.** B.1.2: first sweep for $\frac{\partial z}{\partial x_1}$

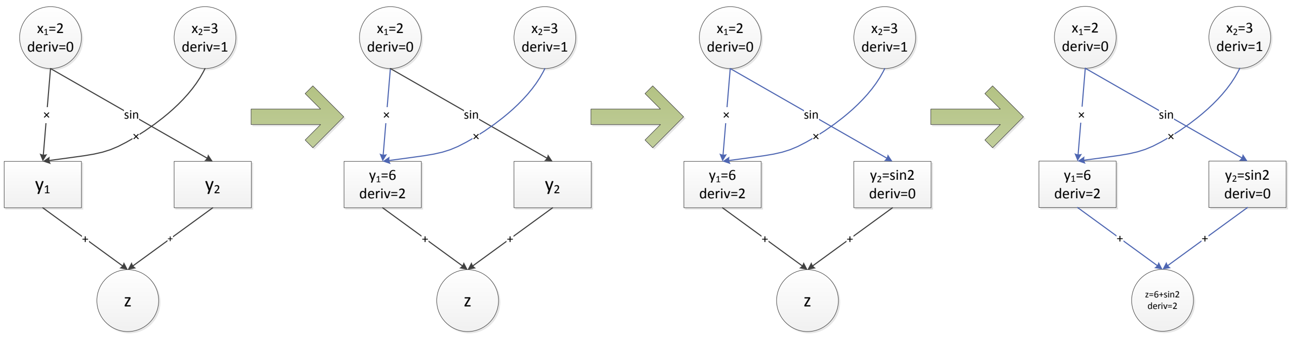


Figure B.6: Forward mode of **Example.** B.1.2: second sweep for $\frac{\partial z}{\partial x_2}$

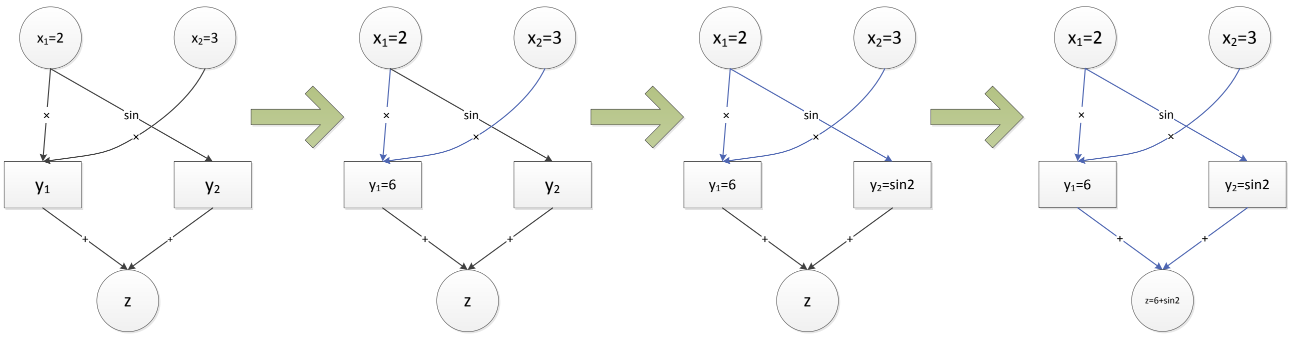


Figure B.7: Reverse mode of **Example.** B.1.2: calculation of values

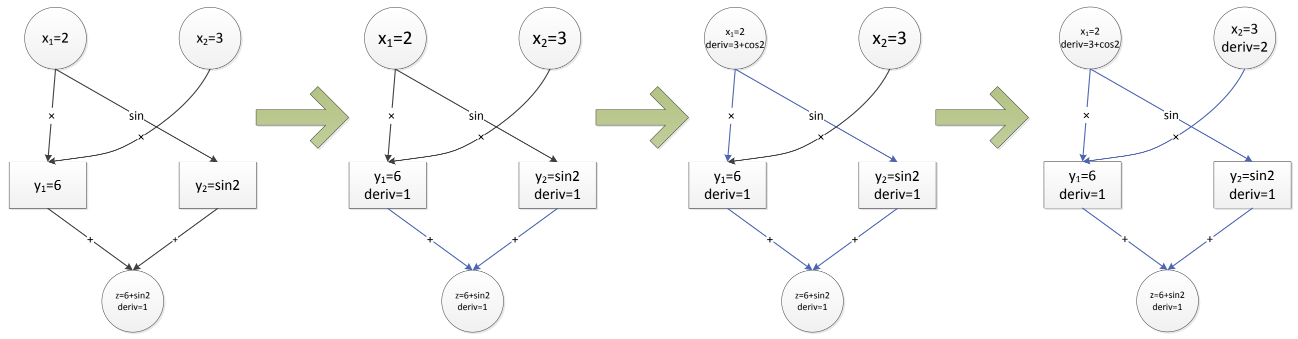


Figure B.8: Reverse mode of **Example.** B.1.2: calculation of *deriv*

Only one reverse sweep is needed since there is only one output variable z. The value of *deriv* at variable z should be set to 1 before starting the computation as shown in **Figure.** B.8.

The gradient can be extracted from *deriv* of input variables x_1 and x_2. The final result is $\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 3 + \cos 2 \\ 2 \end{bmatrix}$, which is identical to that from forward mode.

Cost Analysis

Consider an arbitrary nonlinear function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$. If forward mode ($FM$) is used to calculate the Jacobian matrix, then number of forward sweeps (FS) required is n. The amount of computation for each forward sweep is proportion to that needed to evaluate F is: $\omega(FS(F)) \propto \omega(F)$. Hence the overall all computational cost is:

$$\omega(FM(F)) = n \cdot \omega(FS(F)) \propto n \cdot \omega(F) \tag{B.3}$$

One may notice this method has the same cost as finite difference. Besides no estimation is introduced hence this method is more accurate.

For the same function F, reverse mode (RM) needs one forward sweep to construct tape and m reverse sweeps (RS) to calculate Jacobian matrix. Similar to forward sweep, reverse sweep goes through entire computation once: $\omega(RS(F)) \propto \omega(F)$. Hence we have:

$$\omega(RM(F)) = n \cdot \omega(RS(F)) \propto m \cdot \omega(F) \tag{B.4}$$

Since the cost is proportional to m, not n. The cost can be greatly reduced using reverse mode when $n \gg m$. Furthermore, reverse mode has machine level precision like forward mode. However reverse mode need to record down the tape, which can be large for complicated computations. The main purpose of my work is to reduce the space requirement of reverse mode, and in some cases improve the efficiency at the same time.

B.2 Vector mode of AD

In scalar mode of AD, in the computational graph each node corresponds to a scalar. On contrary, vector mode allows a node to contain a vector or a matrix. In vector mode, since each node holds more information, the length of tape can be much shorter to store for a same computation. Meanwhile number of sweeps required can be reduced for both forward and reverse mode. Consider previous function in **Example.** B.1.2:

Example B.2.1. *AD computation for* $F : \mathbb{R}^2 \mapsto \mathbb{R}$

$$z = F\Big(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\Big) = x_1 \cdot x_2 + \sin x_1 \tag{B.5}$$

Notice its basic operations form is:

$$\begin{cases} y_1 &= & x_1 \cdot x_2 \\ y_2 &= & \sin x_1 \\ z &= & y_1 + y_2 \end{cases}$$

The process of forward mode can be described using **Figure.** B.9.

In vector mode, all dependent variables can be combined into one node, the very first node on the computational graph. Using vector mode the *deriv* attribute of an arbitrary node y_i would record $\frac{\partial y_i}{\partial x}$, its derivative with respect to all depend variables. Notice $\frac{\partial y_i}{\partial x}$ might not be a scalar. It can be a scalar, a vector, a matrix, or a higher dimension matrix, depending on the size of x and y_i. In forward mode, *deriv* at x is initialized to be the identity matrix $I \in \mathbb{R}^{n \times n}$, where $n = 2$ in the above example.

The changes of reverse mode are similar to those of forward mode. Like in scalar mode, it needs a forward sweep first to construct the tape, followed by a reverse sweep to compute the gradient. In the reverse sweep, for an arbitrary node y_i, its *deriv* records $\frac{\partial z}{\partial y_i}$, the derivative of z with respect to the current node y_i. The derivative computation starts at z, at which the *deriv* attribute is initialized to be the identity matrix $I \in \mathbb{R}^{m \times m}$:

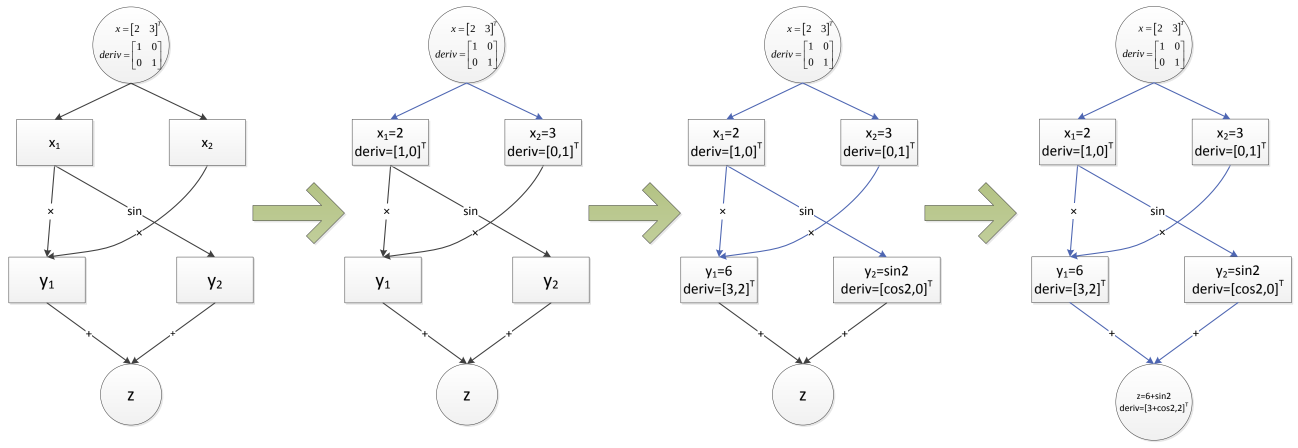


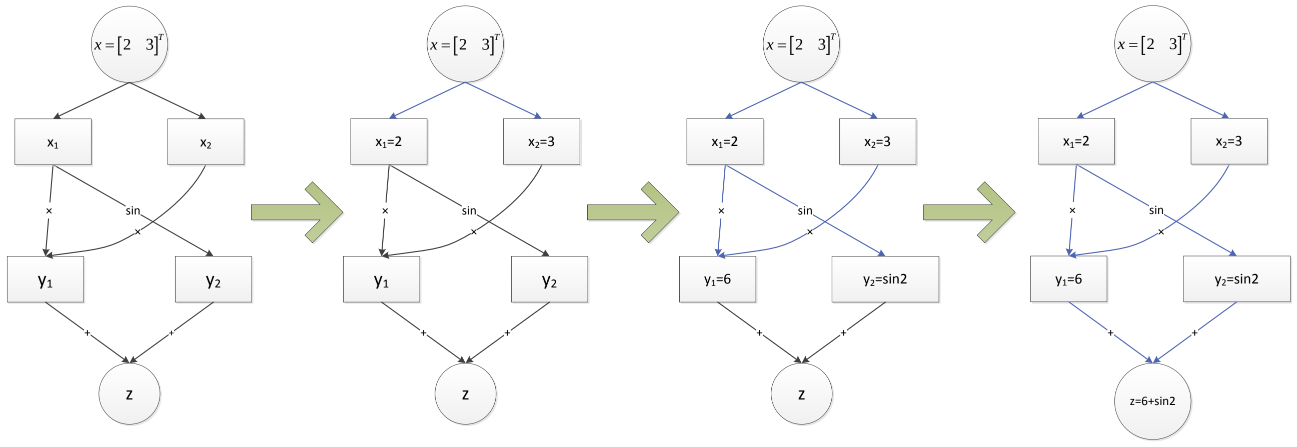
Figure B.9: Vector Mode: Foward mode of **Example.** B.2.1

Cost Analysis

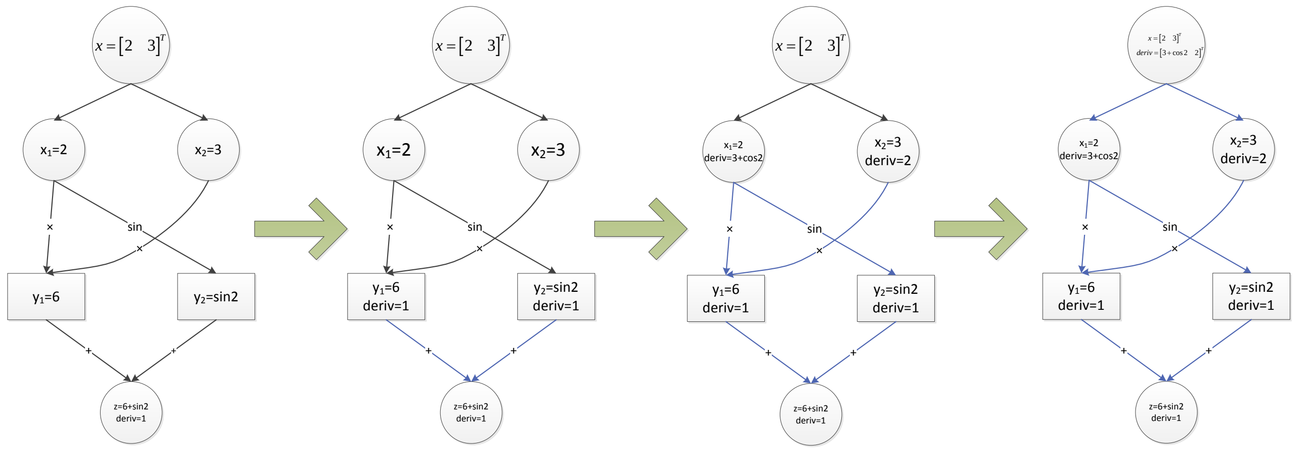
Compared to scalar mode, vector mode in general reduces number of sweeps required. For an arbitrary function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, for forward mode only one forward sweep is required, while n forward sweeps are required in scalar mode. For reverse mode, only one forward sweep and one reverse sweep are required in vector mode, while one forward sweep and m reverse sweep are required in scalar mode. Notice that though the number of sweeps is reduced, the overall computation cost remains the same, because each sweep in vector mode involves matrix operations, which cancel out the effect of fewer sweeps. However, in practice, vector mode is much more efficient in matrix-friendly environment (i.e. MATLAB), because in these environments matrix operations are automatically optimized. It also makes the implementation of AD and the tape more neat and understandable. My work are mostly done under ADMAT-2.0, a MATLAB based AD package. All programs are written in vector mode if not particularly mentioned.

B.3 A Brief Introduction to ADMAT

ADMAT is a Matlab based software which use automatic differentiation idea to compute functions' derivatives, Jacobian matrix, Hessian matrix fast and accurately. While doing computation, there are two modes: forward mode and reverse mode. Two modes' perfor-



(a) Phase 1: calculation of values



(b) Phase 2: calculation of *deriv*

Figure B.10: Vector mode of reverse mode computation of **Example.** B.2.1

mance depends on structure of Jacobian matrix, however for most functions a combination of forward and backward mode works best. A crucial difference between two modes is backward mode needs to record whole computation while forward mode does not. In ADMAT, a global variable ‘tape’ will be created and updated as the record of all executed computations when reverse mode is used.

In matlab, ‘tape’ is a big vector of ‘struct’s, an user defined data type. We call these ‘struct’s cells. Usually each cell corresponds to one basic operation, i.e., plus, times, sin, etc.. Cells in tape are ordered according to the execution time of their respective operations. Each cell owns several child blocks recording type of operation, input cell, constants, and other related information.

B.4 Tape to Graph

To construct the computational graph from a tape, basically one needs to read cells one by one. Typicaly, one cell will be converted into one node. There are also some exceptions, one type of them affects computational graph a lot. In ADMAT, vector operations are also treated as basic operations, i.e., $x_1 + x_2$ where $x_1, x_2 \in \mathbb{R}^{10}$. This operation only occupies one cell in tape, however in the fundamental computational graph it should correspond to 10 output nodes and 20 newly added edges. We are now trying to adjust weights to balance this shape shift, i.e., give these 2 edges heavier weights before applying Ford Fulkerson algorithm.

B.5 Compensation to Condensed Nodes

In fundamental computational graph, we treat each node the same. But in real computational graphs there might be condensed nodes, making it not reasonable. To compensate the distortion, we introduce concept computation intensity to each node. In chaper 3.1 diving graph evenly in fact is in terms of work. Consider equation (2.7), we want $\omega(F_1)$ and $\omega(F_2)$ roughly equal. Therefore if we correctly define computational intensity/work I to each node for the condensed computational graph, then we want a cut such that two subgraphs' computational intensity are roughly the same.

The compensation idea is simple. Originally when evaluating depth for nodes, we try to find length of shortest path from it to any end-node. Here we make a small modification: we still try to find length of shortest path, but redefine length of a path to be sum of its nodes' computational intensity. All later procedures, i.e. evaluation of edges' weights and Ford Fulkerson algorithm, remain the same.

If length of a path is defined in this way, edges with roughly same total intensity on two sides will have the biggest depth and lowest weights. These nodes will be likely to lie in the cut, to divide the graph equally in terms of I. Though no numerical experiment is done yet, we believe this idea is valid.

B.6 The Method to Identify the Intermediate Variables

If the global approach is used to locate the edge separators, to identifiy the intermediate variables is trivial. Simply investigate all the edges going across subgraphs, and the intermediate variables will be the tail nodes of these edges.

If the online approach is used, to identify the intermediate variables is tricky. Since in online approach the graph is never complete, the technique in the global approach is not applicable.

Before the method is formally described, let us first review how the normal computations are done in computers. During the computations of functions, variables are created and destroyed. Each variable has a life time – it is created at some point time, and is destroyed later. It depends on specific functions, but usually a big portion of variables have short life time – they are of temporary use, and hence destroyed shortly after being created. Each variable is destroyed only when it is no longer needed for later computations. Notice that this process is handled automatically by the computer, implying that according to the computer code the computer is able to determine whether a variable is needed again or not in later computations.

As indicated in the previous sections, AD keeps two copies for each variable: 1) the original copy and 2) the copy in the tape. Similar to normal computations, the original copies are created and destroyed automatically. We only need to track the intermediate variables in the tape. The original copy of a variable is destroyed only when it will not be used again in later computations. Therefore every time the original copy of a variable is destroyed, we can place a 'non-intermediate flag' on the associated copy in the tape, and those variables without such flags are the intermediate variable. Since AD uses user-defined classes for the storage and the computations of the original copies of the variables, by inserting the 'place-flag operation' into the destructors of the user-defined classes, the intermediate/non-intermediate variables in the tape can be tracked automatically.

References

[1] Sanjeev Arora, Elad Hazan, and Satyen Kale. $o(\sqrt{\log n})$ approximation to sparsest cut in $\tilde{O}(n^2)$ time. *SIAM J. Comput.*, 39(5):1748–1771, jan 2010.

[2] Brett M. Averick, Jorge J. Moré, Christian H. Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 15(2):285–294, 1994.

[3] Christian H. Bischof, Ali Bouaricha, Peyvand Khademi, and Jorge J. Moré. Computing gradients in large-scale optimization using automatic differentiation. *INFORMS J. Computing*, 9:185–194, 1997.

[4] Christian H. Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[5] Christian H. Bischof and Mohammad R. Haghighat. Hierarchical approaches to automatic differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94. SIAM, Philadelphia, PA, 1996.

[6] Christian H. Bischof, Peyvand M. Khademi, A. Bouaricha, and Alan Carle. Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7:1–39, 1997.

[7] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[8] H. Martin Bücker and A. Rasch. Modeling the performance of interface contraction. *ACM Transactions on Mathematical Software*, 29(4):440–457, 2003.

[9] Luca Capriotti and Michael B. Giles. Algorithmic differentiation: Adjoint greeks made easy. *Computational Finance*, 2011.

[10] Thomas F. Coleman and Jin-Yi Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7(2):221–235, 1986.

[11] Thomas F. Coleman and Gudbjorn F. Jonsson. The efficient computation of structured gradients using automatic differentiation. *SIAM Journal on Scientific Computing*, 20(4):1430–1437, 1999.

[12] Thomas F. Coleman, Fadil Santosa, and Arun Verma. Semi-automatic differentiation. In Jeff Borggaard, John Burns, Eugene Cliff, and Scott Schreck, editors, *Computational Methods for Optimal Design and Control*, volume 24 of *Progress in Systems and Control Theory*, pages 113–126. Birkhuser Boston, 1998.

[13] Thomas F. Coleman, Fadil Santosa, and Arun Verma. Efficient calculation of Jacobian and adjoint vector products in wave propagational inverse problem using automatic differentiation. *J. Comp. Phys.*, 157:234–255, 2000.

[14] Thomas F. Coleman and Arun Verma. Structure and efficient Jacobian calculation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 149–159. SIAM, Philadelphia, PA, 1996.

[15] Thomas F. Coleman and Arun Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, 1998.

[16] Thomas F. Coleman and Arun Verma. Structure and efficient Hessian calculation. In Ya-Xiang Yuan, editor, *Proceedings of the 1996 International Conference on Nonlinear Programming*, pages 57–72. Kluwer Academic Publishers, 1998.

[17] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software*, 26(1):150–175, 2000.

[18] Thomas F. Coleman and Xin Xiong. New graph approaches to the determination of Jacobian and Hessian matrices, and Newton steps, via automatic differentiation (in preparation).

[19] Thomas F. Coleman and Wei Xu. Fast (structured) Newton computations. *SIAM Journal on Scientific Computing*, 31(2):1175–1191, 2008.

[20] Thomas F. Coleman and Wei Xu. The efficient evaluation of structured gradients (and underdetermined Jacobian matrices) by automatic differentiation, 2013.

[21] Alexander W. Dowling, Sree R. R. Vetukuri, and Lorenz T. Biegler. Large-scale optimization strategies for pressure swing adsorption cycle synthesis. *AIChE Journal*, 58(12):3777–3791, 2012.

[22] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[23] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[24] Assefaw Hadish Gebremedhin, Arijit Tarafdar, Fredrik Manne, and Alex Pothen. New acyclic and star coloring algorithms with applications to Hessian computation. *SIAM Journal on Scientific Computing*, 29(3):1042–1072, 2007.

[25] Andreas Griewank. Direct calculation of Newton steps without accumulating Jacobians. In Thomas F. Coleman and Yuying Li, editors, *Large-Scale Numerical Optimization*, pages 115–137. SIAM, Philadelphia, Penn., 1990.

[26] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[27] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.

[28] Andreas Griewank, David Juedes, and Jean Utke. A package for the automatic differentiation of algorithms written in C/C++. User manual. Technical report, Institute of Scientific Computing, Technical University of Dresden, Dresden, Germany, 1996. this version of the manual is superceded by http://www.math.tu-dresden.de/~adol-c/adolc110.ps.

[29] A.K.M. Shahadat Hossain and Trond Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[30] Shahadat Hossain and Trond Steihaug. Sparsity issues in the computation of Jacobian matrices. In Teo Mora, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC)*, pages 123–130, New York, NY, 2002. ACM.

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

[32] David E. Keyes, Paul D. Hovland, Lois C. McInnes, and Widodo Samyono. Using automatic differentiation for second-order matrix-free methods in PDE-constrained optimization. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 3, pages 35–50. Springer, New York, NY, 2002.

[33] Claire Lauvernet, Laurent Hascot, Franois-Xavier Dimet, and Frdric Baret. Using automatic differentiation to study the sensitivity of a crop model. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 59–69. Springer Berlin Heidelberg, 2012.

[34] Johannes Lotz, Uwe Naumann, and Jrn Ungermann. Hierarchical algorithmic differentiation a case study. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 187–196. Springer Berlin Heidelberg, 2012.

[35] Azamat Mametjanov, Boyana Norris, Xiaoyan Zeng, Beth Drewniak, Jean Utke, Mihai Anitescu, and Paul Hovland. Applying automatic differentiation to the community land model. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 47–57. Springer Berlin Heidelberg, 2012.

[36] P. M. Pardalos and Andreas Griewank. Some bounds on the complexity of gradients, jacobians, and hessians, 1993.

[37] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.

[38] DanielR. Reynolds and Ravi Samtaney. Sparse jacobian construction for mapped grid visco-resistive magnetohydrodynamics. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 11–21. Springer Berlin Heidelberg, 2012.

[39] E. M. Tadjouddine. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688–699, 2008.

[40] Wei Xu and Thomas F. Coleman. Efficient (partial) determination of derivative matrices via automatic differentiation (to appear in SIAM journal on Scientific Computing, 2012).