On the Use of Directed Cutsets to Reveal Structure for Efficient Automatic Differentiation of Multivariate Nonlinear Functions

by

Xin Xiong

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Thomas F. Coleman

Waterloo, Ontario, Canada, 2011

Abstract

This paper is concerned with the efficient computation of Jacobian matrices of nonlinear vector maps using automatic differentiation (AD). Specificially, we propose the use of a directed cutset method, weighted minimum cut, 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 discuss the results of numerical experiments significant practical potential of this approach.

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 Figures **vii**

1 Introduction **1**

1.1 Automatic Differentiation and The Cutset 2

1.2 Automatic Differentiation and Multiple Cutsets 7

2 On Finding Cutsets to Increase Efficiency in the Application of Automatic Differentiation **10**

2.1 Weighted Minimum Cut 10

3 Experiments **14**

3.1 Thin Computational Graphs 14

3.2 Fat Computational Graphs 15

4 Accelerating the Calculation of the Jacobian matrix **18**

5 Concluding Remarks **20**

APPENDICES **22**

A Sparsest Cut **23**

B Generation of Computational Graphs **24**

B.1 A Brief Introduction to ADMAT 24

B.2 Tape to Graph 24

B.3 Compensation to Condensed Nodes 26

References **27**

List of Figures

1.1 Head node and tail node of a given edge e_k 4
1.2 An example of computational graphs and a sample directed cut 5

2.1 An example of weighted min cut 11
2.2 Depth of edges in F's (equation (1.3)) computational graph 12
2.3 Performance of Ford Fulkerson algorithm on F_k 13

3.1 Weights and obtained cutsets of FF_1's condensed computational graph . . 16
3.2 Weights and obtained cutsets of FF_2's computational graph 17

4.1 Acceleration of cutset method 19

B.1 Equation (1.3)'s computational graph, without merging duplicated nodes . 25

Chapter 1

Introduction

Many scientific and engineering computations require the repeated calculation of matrices of derivatives. 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 $x \in \mathbb{R}^n$, 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 these derivative matrices often represents a significant portion of the overall computational cost of the process. Therefore there is notable value in general methods and technology that yield derivate matrices accurately and efficiently.

Automatic differentiation (AD) is a field, intersecting computer science and applied mathematics, that has advanced rapidly over the past 15 years [2]. AD can deliver matrices of derivatives given a source code to evaluate the function F (or in the case of minimization, the objective function f). Good methods that exploit sparsity, constant values, or duplicate values, have also been developed [3]. In addition, if the objective function exhibits certain kinds of structures, and this structure is conveniently noted in the expression of the objective function, then the efficiency of the automatic differentiation process can be greatly enhanced [2, 4, 5, 9, 10, 11, 14].

This paper is concerned with the case where the problem structure is not noted priori and automatic differentiation may subsequently be regarded as too costly either in time or space. For example, consider the automatic differentiation of the gradient function $\nabla f(x) = \left(\frac{\partial f}{\partial x_i}\right)_{i=1:n} \in \mathbb{R}^n$, given the source code to evaluate the scalar-valued objective function $f(x)$. It is well-known that the reverse-mode of automatic differentiation can be applied to yield the gradient, accurately, in time proportional to the time required to evaluate the function itself. However, the space required to obtain this (theoretical) running time is proportional to the total number of arithmetic operations required to evaluate f, and this can mean that in practice (due to lack of sufficient fast memory), the *realized* running time is considerably (sometimes dramatically) worse [4] than the theory predicts. Other examples, involving the determination of a Newton step for a system of nonlinear equations, are given in [5]. The solution to this space challenge proposed in [5] is to identify and exploit the structure in the problem before applying automatic differentiation and then apply AD in a structured way. This explicit 'slice and dice' approach is effective but does require that the user understand and work with the prescribed notion of problem structure. In this paper we discuss a more general, less intrusive, solution.

1.1 Automatic Differentiation and The Cutset

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: $J_{ij} = \frac{\partial f_i}{\partial x_j}(i = 1, \cdots, m; j = 1, \cdots, n)$. Given the source code to evaluate $F(x)$, automatic differentiation can be used to determine $J(x)$. Generally, the work required to evaluate $J(x)$ via a *combination* of the forward and reverse modes of AD, and in the presence of sparsity in $J(x)$, is propotional 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 basic computational steps) to evaluate the argument -see [10]. We note that when reverse mode AD is invoked the space required to compute the Jacobian is proportional to $\omega(F)$, and this can be prohibitively large. 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. 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 [10].

Lemma 1.1.1. *Assume $f_i : \mathbb{R}^n \mapsto \mathbb{R}^1$ is differentiable and the Jacobian matrix J is computed by the bi-coloring AD method [10]. If assume optimal coloring is found, then in general,*

$$\left.\begin{array}{c} \omega(J) = O(\chi_B(G^D(J)) \cdot \omega(F) + |J|_{NNZ}) \\ \sigma(J) = O(\omega(F) + |J|_{NNZ}) \end{array}\right\} \tag{1.1}$$

Proof. According to [10], 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 $O(|J|_{\text{NNZ}})$ if $W^T J$ and JV are given. We can obtain J in $O(|J|_{\text{NNZ}})$ because at least one none zero entry is determined in one substitution, so at most $|J|_{\text{NNZ}}$ substitutions are requred. 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 $O(t_W \cdot \omega(F))$ [10]. If the optimal coloring is found

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

and then

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

The second equation in (1.1) is obviously true because the reverse mode of AD needs $O(\omega(F))$ space [10] and J itself needs $O(|J|_{\text{NNZ}})$ space. ☐

The bi-coloring AD method does not guarantee to find 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 (1.1).

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

$$\vec{G}(F) = (V, \vec{E}) \tag{1.2}$$

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

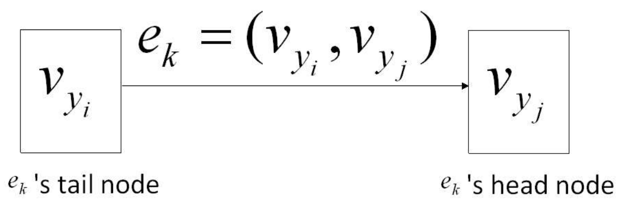


Figure 1.1: Head node and tail node of a given edge e_k

$z_j = [F(x)]_j$ 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 1.1 for an illustration. It is clear that if F is well-defined then $\vec{G}(F)$ is an acyclic graph.

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

$$F\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} \sin(\cos(\sin 2^{x_1} + x_2^2) \cdot (5x_1 - 6x_2)) \\ (2x_1^{x_2} + x_2^{x_1})^{\sin x_1 + \cos x_2} \\ \cos(\sin 2^{x_1} + x_2^2) + (5x_1 - 6x_2) + (2x_1^{x_2} + x_2^{x_1}) + (\sin x_1 + \cos x_2) \end{bmatrix} \tag{1.3}$$

Then F 's computational graph is Fig. 1.2(a):

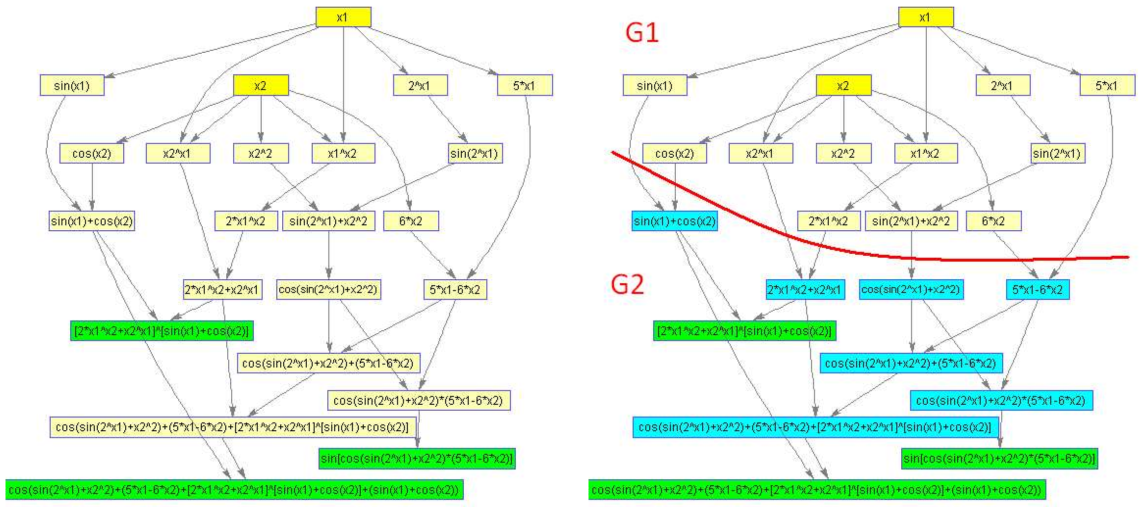
Definition 1.1.2. *$E_{cut} \in \vec{E}$ is a directed* cutset *in directed graph $\vec{G}$ if $\vec{G} - \{E_{cut}\} = \{G_1, G_2\}$ where G_1, G_2 are disjoint and all edges in E_{cut} have the same orientation relative to G_1, G_2.*

Example. One choice of a cut for F defined by equation (1.3) is given in Fig. 1.2(b).

Suppose $E_{cut} \subset \vec{E}_y$ is a cutset of the computational graph $\vec{G}(F)$ with orientation forward in time. Then the nonlinear function $F(x)$ can be broken into two parts:

$$\left.\begin{array}{r} \text{solve for } y\text{: } F_1(x, y) = 0 \\ \text{solve for } z\text{: } F_2(x, y) - z = 0 \end{array}\right\} \tag{1.4}$$

where y is the vector of intermediate variables defined by the *tail vertices* of the edge cutset E_{cut}, and z is the output vector, i.e., $z = F(x)$. Let p be the number of tail vertices


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))
G1
G2

(a) F's computational graph G (b) An example of graph G's directed cut

Figure 1.2: An example of computational graphs and a sample directed cut

of edge set E_{cut}, i.e., $y \in \mathbb{R}^p$. Note: $|E_{cut}| \geq p$. The nonlinear function F_1 is defined by the computational graph above E_{cut}, i.e., G_1, and nonlinear function F_2 is defined by the computational graph below E_{cut}, i.e., G_2. See Figure 1.2(b). We note that the system (1.2) can be differentiated wrt (x, y) to yield an 'extended' Jacobian matrix:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & (F_1)_y \\ (F_2)_x & (F_2)_y \end{bmatrix} \tag{1.5}$$

Since y is a well-defined unique output of function $F_1 : \mathbb{R}^n \mapsto \mathbb{R}^{n+p}$, $(F_1)_y$ is a $p \times p$ non-singular matrix. The Jacobian of F is the Schur-complement of (1.4), i.e.,

$$J(x) = (F_2)_x - (F_2)_y (F_1)_y^{-1} (F_1)_x \tag{1.6}$$

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 1.1.3** formalizes this.

Theorem 1.1.3. *Assume the computational graph G is divided into two disjoint subgraphs G_1, G_2 with the removal of directed cutset E_{cut} as described above. Let J_E be computed by* *the bi-coloring technique [10]. Then assuming optimal graph coloring, in general,*

$$\left.\begin{array}{rcc} \omega(J_E) & = & O(\chi_B(G^D[(F_1)_x,(F_1)_y]) \cdot \omega(F_1) + \chi_B(G^D[(F_2)_x,(F_2)_y]) \cdot \omega(F_2) + |J_E|_{NNZ}) \\ \sigma(J_E) & = & O(\max(\sigma[(F_1)_x,(F_1)_y],\sigma[(F_2)_x,(F_2)_y]) + |J_E|_{NNZ}) \\ & = & O(\max(\omega(F_1),\omega(F_2)) + |J_E|_{NNZ}) \end{array}\right\} \tag{1.7}$$

Proof. According to (1.5), to determine J_E, we first determine $\left[(F_1)_x \quad (F_1)_y\right]$, and then determine $\left[(F_2)_x \quad (F_2)_y\right]$. By **Lemma 1.1.1**,

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

Now consider the space: To determine $\left[(F_1)_x \quad (F_1)_y\right]$, $O(\omega(F_1))$ is required for the reverse mode of AD. Then when start to evaluate $\left[(F_2)_x \quad (F_2)_y\right]$, previous memory can be clear and needs $O(\omega(F_2))$ space for F_2's reverse mode. Consider the whole progress, total space requirement is the peak usage which is $O(\max(\omega(F_1),\omega(F_1)))$. Besides extra $O(|J_E|_{\text{NNZ}})$ space is needed for the restore results obtained, and hence

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

□

We can compare (1.7) to (1.1) to contrast the time/space requirements of the cutset/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: (1.7, $\sigma(J_E)$) indicates that the required space is about $O(\max(\omega(F_1),\omega(F_2)))$, 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 (1.1). For example, if the cutset 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 (1.7,$\omega(J_E)$) to (1.1,$\omega(J)$) can either increase or decrease. However, due to increased sparsity

$$\begin{array}{c} \chi_B(G^D[(F_1)_x,(F_1)_y]) \leqslant \chi_B(G^D(J)) \\ \chi_B(G^D[(F_2)_x,(F_2)_y]) \leqslant \chi_B(G^D(J)) \end{array} \tag{1.8}$$

and then

$$\begin{array}{rcl}\chi_B(G^D[(F_1)_x,(F_1)_y])\cdot\omega(F_1)+\chi_B(G^D[(F_2)_x,(F_2)_y])\cdot\omega(F_2) & \leqslant & \chi_B(G^D)\cdot(\omega(F_1)+\omega(F_2)) \\ & = & \chi_B(G^D)\cdot\omega(F)\end{array} \tag{1.9}$$

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

We have shown above that it can be 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. However, it is reasonable to ask: 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

$$\text{solve } J_E\begin{bmatrix} s \\ t\end{bmatrix} = \begin{bmatrix} 0 \\ -F\end{bmatrix} \tag{1.10}$$

the main points are that calculating matrix J_E can be less costly than calculating matrix J, and solving (1.10) can also be relatively inexpensive given sparsity that can occur in J_E that may not be present in J.

1.2 Automatic Differentiation and Multiple Cutsets

The ideas discussed above can be generalized to the case with multiple mutually independent edge cutsets, $E_{cut_1}, \cdots, E_{cut_k} \in \vec{E}$, where we assume $G - \{E_{cut_1}, \cdots, E_{cut_k}\} = \{G_1, \cdots, G_{k+1}\}$. The 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 $G_{i+1}, i = 1 : k$

Suppose $E_{cut_1}, \cdots, E_{cut_k} \in \vec{E}$ are pairwise disjoint cutsets of the computational graph $\vec{G}(F)$ with orientation forward in time (as indicated above). Then the nonlinear function $F(x)$ can be broken into $k+1$ parts:

$$\left.\begin{array}{rcc}\text{solve for } y_1 & : & F_1(x,y_1)=0 \\ \text{solve for } y_2 & : & F_2(x,y_1,y_2)=0 \\ \vdots & & \vdots \\ \text{solve for } y_k & : & F_k(x,y_1,\cdots,y_k)=0 \\ \text{solve for } z & : & F_{k+1}(x,y_1,\cdots,y_k)-z=0\end{array}\right\} \tag{1.11}$$

where y_i is the vector of intermediate variables defined by the *tail vertices* of the edge cutset E_{cut_i}, for $i = 1, \cdots, k+1$ and z is the output vector, i.e., $z = F(x)$. Let p_i be the number of tail vertices of edge set E_{cut_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_{cut_i}, i.e., G_i. We note that the system (1.11) can be differentiated wrt (x, y) to yield an 'extended' Jacobian matrix:

$$J_E \triangleq \begin{bmatrix} (F_1)_x & (F_1)_{y_1} & 0 & 0 & 0 & 0 \\ (F_2)_x & (F_2)_{y_1} & (F_2)_{y_2} & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & . & \vdots \\ (F_k)_x & (F_k)_{y_1} & (F_k)_{y_2} & \cdots & \cdots & (F_k)_{y_k} \\ (F_{k+1})_x & (F_{k+1})_{y_1} & (F_{k+1})_{y_2} & \cdots & \cdots & (F_{k+1})_{y_k} \end{bmatrix} \tag{1.12}$$

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 $(F_i)_{y_j}$ is a $p_i \times p_i$ non-singular 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-cut 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

$$J_E \begin{bmatrix} s \\ t_1 \\ \vdots \\ t_k \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ -F \end{bmatrix} \tag{1.13}$$

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

$$J_E = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{1.14}$$

where B is nonsingular, and

$$J = C - DB^{-1}A \tag{1.15}$$

The space/time requirements for the multi-cutset case will be formalized in future work, in analogy to the 1-cut case captured by **Theorem 1.1.3**.

Chapter 2

On Finding Cutsets to Increase Efficiency in the Application of Automatic Differentiation

In Section 1.1 we observed that if a small directed cut divides the computational graph G into roughly two equal components G_1 and G_2, then the space requirement are minimized (roughly halved). Moreover, the required work, as indicated by equation (1.7), will not increase, and due to inreasing sparsity, will likely decrease.

Therefore, our approach will be to seek a small directed cutset that will (roughly) bisect the fundamental computational graph.

2.1 Weighted Minimum Cut

Our intial weighted min cut method is based on the Ford Fulkerson algorithm [6], a well known max-flow/min-cut algorithm. Ford Fulkerson algorithm automatically finds the minimum directed $s-t$ cut. An $s-t$ cut is set of edges whose removal seperates specified node s and node t, two arbitary nodes in the graph. Minimum cut means this cut has smallest capacity among all possible cuts, where a cut's capacity is defined to be sum of all capacities of forward edges in it. See Figure 2.1 for an example of weighted min cut. Numbers on edges are the flow and capacity. i.e. 1/7 means that egdes has flow 1 and capacity 7. Notice the minimum cut has full flows on all its forward edges.

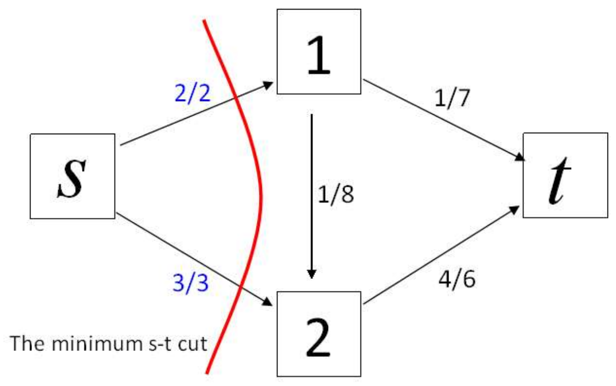


Figure 2.1: An example of weighted min cut

If a capacity of '1' is assigned to every edge, Ford Fulkerson will find the smallest cutset. However, as mentioned above, instead of merely finding a minimum cut, we also desire that the determined cut (roughtly) divide the fundamental computational graph in half. To add this preference into the optimization, we assign different capacities to different edges such that away-ends-edges, i.e., edges far away from input nodes and output nodes, have smaller capacities, while near-ends-edges have bigger capacities. With this kind of nonuniform distribution, a 'small' cut will likely be loacted towards the middle of the fundamental computaional graph.

In practice we first calculate depth of edges. See Def. 2.1.1 for definition of an edge's depth. See Figure 2.2 for an example.

Definition 2.1.1. *Depth of an edge is the length of shortest undirected path reaching a input node or a output node starting from this edge.*

Example. Figure 2.2 is depth of edges in F's computational graph, where F is defined by equation (1.3).

Then define a decreasing function

$$f : \mathbb{Z}^+ \mapsto \mathbb{Z}^+ \tag{2.1}$$

taking depth as inputs and gives capacities for each edge as outputs. Since capacities is an evaluation of influence/importance of edges, we also call them weights. Notice weights are restricted to be integers because Ford Fulkerson algorithm runs faster for integral edge capacities. There are many choices of f. Which one to choose in fact depends on and

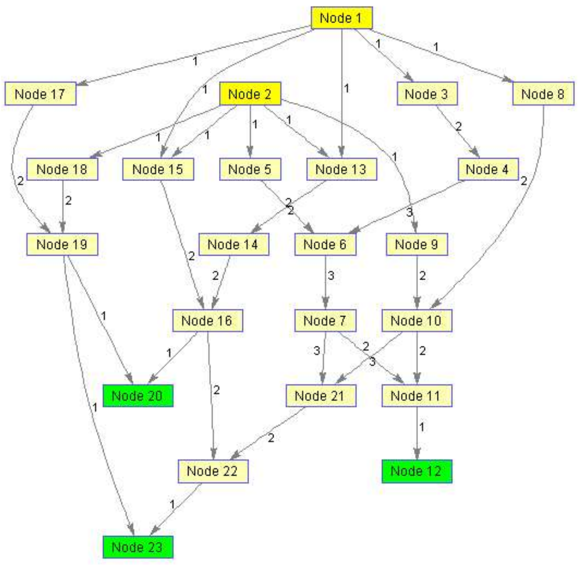


Figure 2.2: Depth of edges in F's (equation (1.3)) computational graph

performance of numerical experiments heuristics. f may even depend on the function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$. Currently find that quadratic weights give reasonable results. See Chapter 3 for numerical examples. Once weights are determined, Ford Fulkerson algorithm can be applied and cutsets can be obtained.

Cost Analysis

For integral capacities, run time of Ford-Fulkerson is bounded by $O(|E|f)$ [6] where $|E|$ is number of edges in the graph, f is the maximum flow. In computational graphs, equation (2.1) is alawys a decreasing function so weights in the middle are smaller. Due to this special distribution, usually there exists a cut in the middle with a small capacity, 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. Use function defined in equation (4.1) 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 2.3.

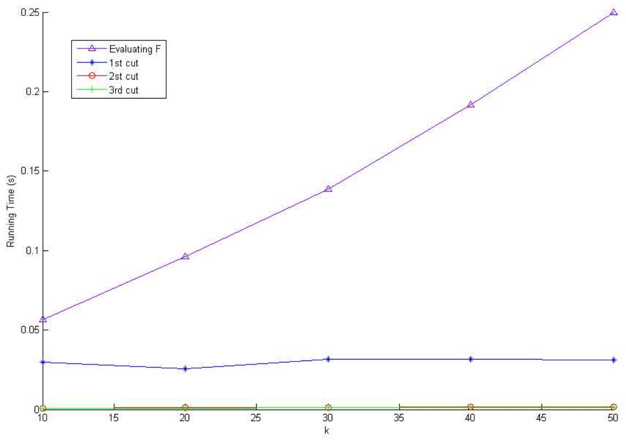


Figure 2.3: Performance of Ford Fulkerson algorithm on F_k

Multiple Cutsets

For now there is a simple construction of multiple cutsets finding scheme. Apply single cutset finding scheme recursively by doing the following: Always record size of all subgraphs. Every time after a new cutset is found, apply the single cutset finding scheme again on the biggest subgraph, until enough number of cutsets are generated.

Chapter 3

Experiments

In this chapter we provide computational results on some preliminary experiments to automatically reveal 'AD-useful' structure using the cutset idea. These initial experiments are based on weighted min-cut algorithms where edge weights are chosen, in a very simple preliminary way, to find cuts that bisect the fundamental computational graph.

In particular, we weigh the edges, based on the fundamental computational graph, as follows:

$$W_{e_i} = (\max_j D_{e_j} - D_{e_i})^2 + 1; \tag{3.1}$$

where depth D_e is calculated by Def. 2.1.1.

We use the AD-tool, ADMAT(Appendix B.1) to generate the computational graphs. However, for efficiency reasons, ADMAT sometimes condenses 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 must be adjusted to account for this. (Appendix B.3)

In our numerical experiments we focus on two types of structure that represent the two shape extreme cases. Specially, we consider long thin computational graphs, and short fat graphs.

3.1 Thin Computational Graphs

Usually a thin computational graph involves iterations where result in one iteration is used as initial value in the next iteration.

Example. If define

$$F_1\left(\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right)=\begin{bmatrix}x_3\cdot\cos(\sin(2^{x_1}+x_2^2))\\5x_1-6x_2\\2x_2^{x_2}+x_2^{x_1}\end{bmatrix}\tag{3.2}$$

and

$$FF_1=F_1\circ F_1\circ F_1\circ F_1\circ F_1\circ F_1$$

FF_1's computational graph is thin.

Weights used to find the first cutset are shown is Figure 3.1(a). After three interations, three cutsets in Figure 3.1(b) are found. Graph is divide into four subgraphs.

Visually, these cuts are good in terms of size and evenly dividing the graph.

3.2 Fat Computational Graphs

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

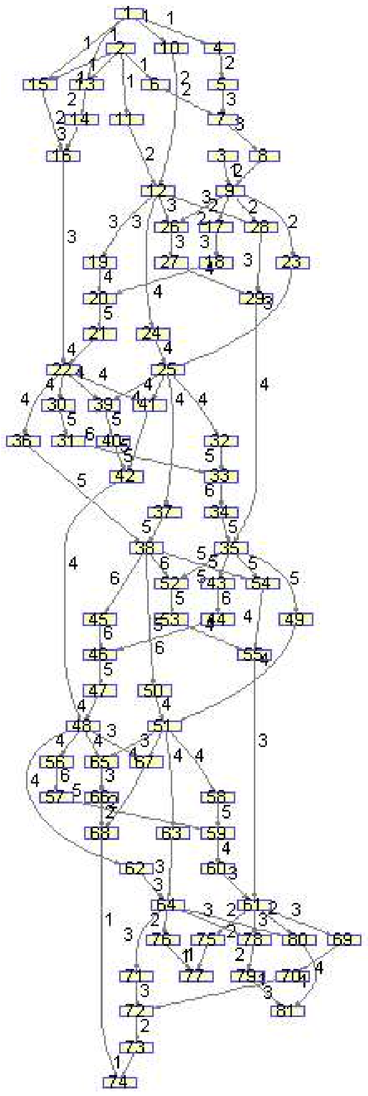
$$FF_2=\sum_{i=1}^{6}F_1(x+\mathrm{rand}_i(3,1))$$

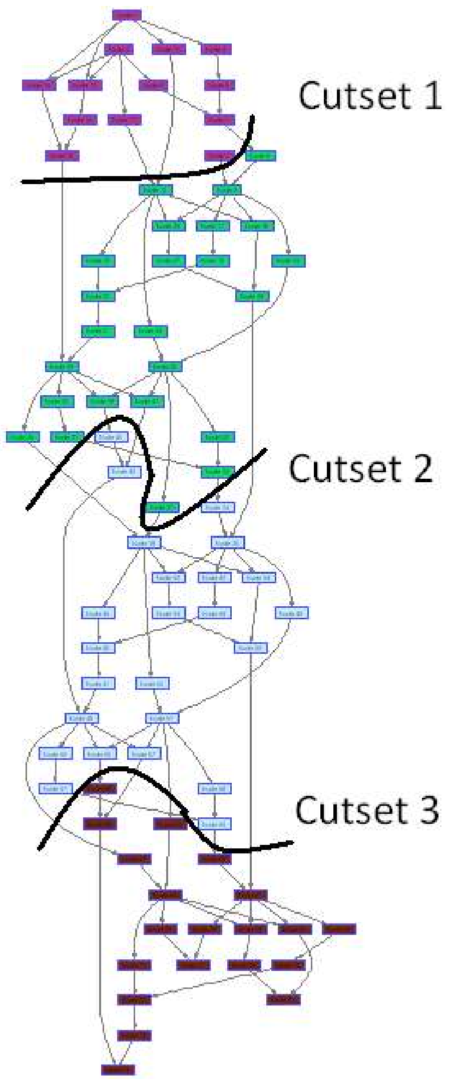
where F_1 is defined by eqution (3.2) in the previous experiment.

Weights follow equation (3.1), and they are shown graphically in Figure 3.2(a).

Cut indicated in Figure 3.2(b) are found by algorithm stated before. However we wish to find a cut indicated by the horizontal curve, which is very different from the real cut located. This is mainly because of condensed nodes. In this particular example, each node under the curve represents a vector of dimension four. In its fundamental computational graph, each of these nodes is supposed to split into four nodes, making the two subgraphs separated by the curve more balanced.

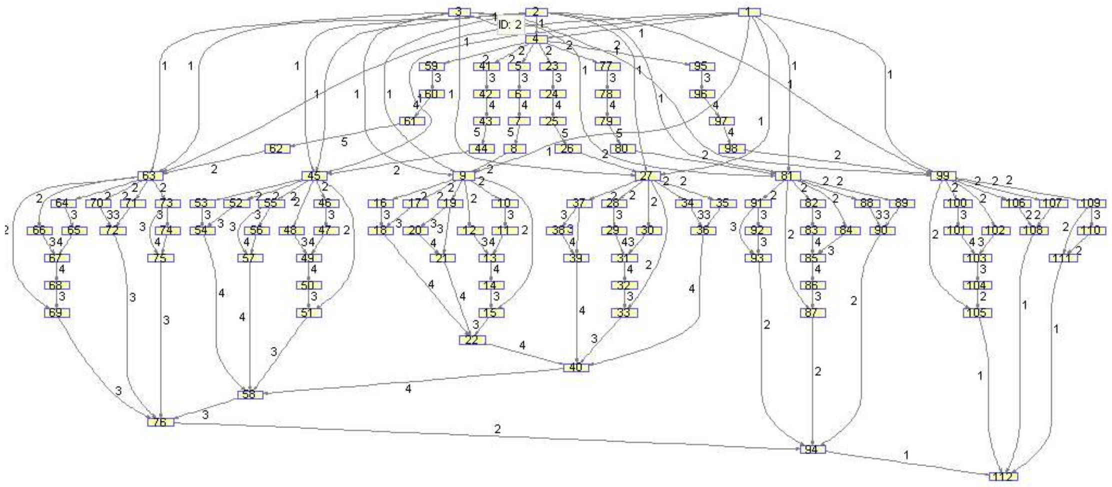
Weighted min cut does not work very well on fat computational graphs. Slight asymmetry also leads to disturbance of cut's distribution. For this kind of functions, we need to do re-weighting(Appendix B.3) or even explore other better algorithms to analyse their structure.

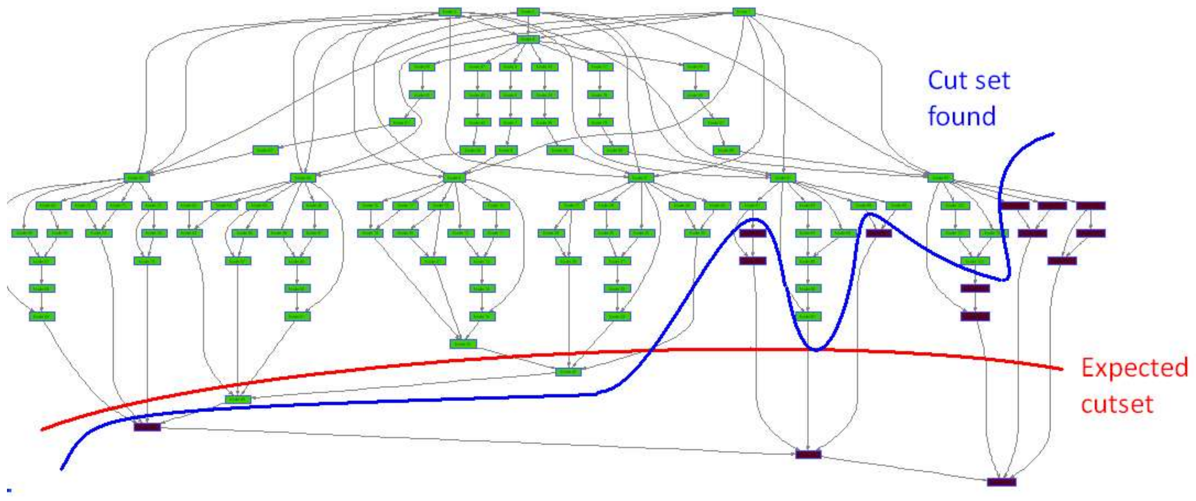

(a) Weights of Edges



(b) Cutsets Founded

Figure 3.1: Weights and obtained cutsets of FF_1's condensed computational graph


(a) Weights of Edges



(b) Cutsets Founded

Figure 3.2: Weights and obtained cutsets of FF_2's computational graph

Chapter 4

Accelerating the Calculation of the Jacobian matrix

To illustrate how cutsets 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

$$F_k = f \circ f \circ \cdots \circ f \qquad \text{where there are } k \ f\text{'s} \tag{4.1}$$

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 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 reverse mode to obtain J both directly and by constructing cutsets. Their running time and space usage will be recorded to see improvements. If cutset method is used, J_E defined by equation (1.12) will be calculated, and J will be further calculated by equation (1.14) and (1.15).

By doing experiment, the Figure 4.1 is obtained.

For cases that cutsets method is introduced, computational graph is always divided evenly. For example: In one cut case, F_{2400} is treated as $F_{1200} \circ F_{1200}$. In two cuts 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 1.1.3**. This is a quite remarkable result.

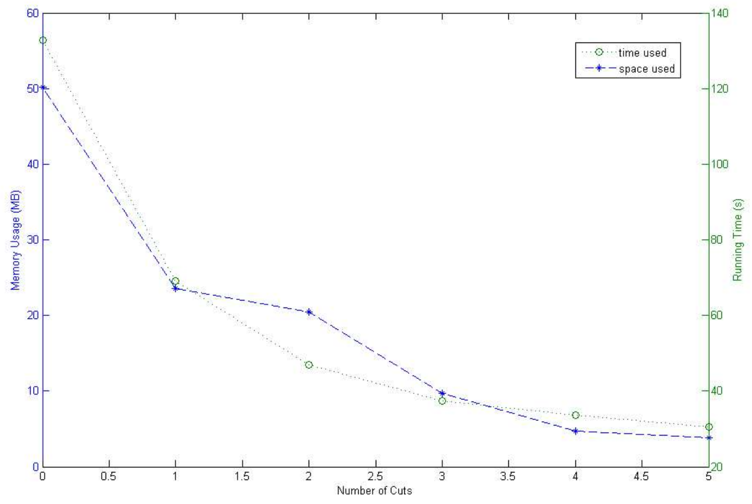


Figure 4.1: Acceleration of cutset method

The performance plot in Figure 4.1 did not count in time used to locate cutsets. The 'running time' refers to the time used to obtain Jacobian matrix with cutsets provided. In practice, generation of computational graphs and analysis may be costly, compared with those used to get Jacobian matrix direclty. However once the desired cutsets 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 cutset method need only be applied a single time.

The computational graph of F_k is a long thin graph. Our method locates small cutsets that tend to break the graph in a well-balanced way. So cutsets optimization is expected to have good performance. In practice, computational graphs are not so ideal; hence running time and memory may not be reduced so dramatically, but we quite expect a significant improvement.

Chapter 5

Concluding Remarks

Our initial experiments and analysis indicate that separation of nonlinear systems with use of directed cutsets can significantly reduce the computing time. Continued research along these lines is recommended. Issues to be further investigated include:

- Fat computational graphs pose challenges to our current edge-weighting scheme. More research is needed here.
- We have not focused on the expense of locating the cutsets. Note that this operation need only to be computed once for a nonlinear system, this cost is amortized over many iterations. Nevertheless, research on cutset efficiency 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.
- To reduce memory usage when generate computational graph, one possible way is to use an online algorithm, which refers to generation of cuts with only partial information. i.e. When evaluating a function F, only a maximum of 100MB information is recorded. As calculation goes on, old information are deleted before new operations

are recorded, making sure overall memory usage never exceed 100MB. Instead of waiting for the whole graph, we always generate cutsets from the partial graph. We wish to develop this idea in future work.

APPENDICES

Appendix A

Sparsest Cut

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

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

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 [7].

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 [7], 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 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' performance 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.2 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

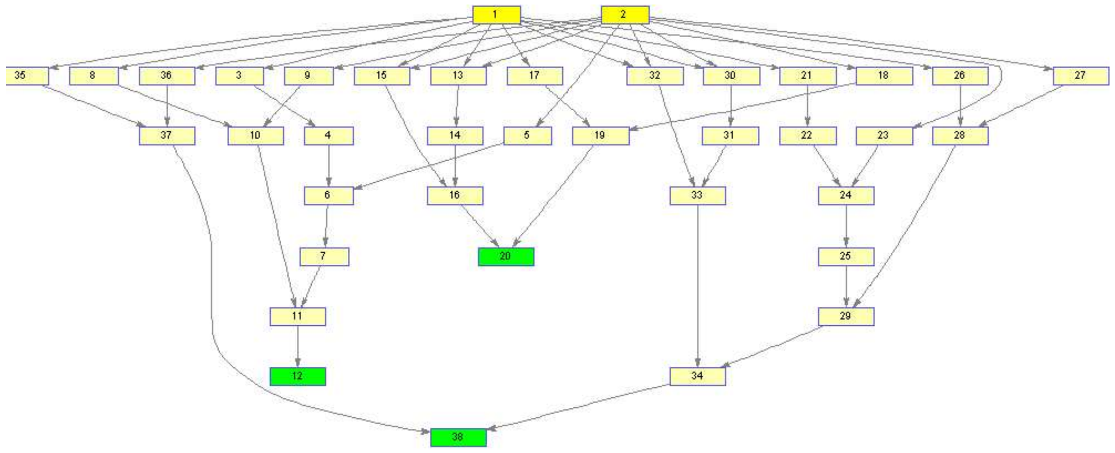


Figure B.1: Equation (1.3)'s computational graph, without merging duplicated nodes

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.

Another extra work need to be done is eliminating duplicated nodes. Because ADMAT is such implemented so that variable repeatedly used does not just occupy one cell. Instead a new cell is always created when a variable is needed, no matter if it is already calculated or not. When computational graph is first generated, it often contains lots of duplicated nodes — different nodes but representing exactly a same variable. So we have to merge them to get the real computational graph.

Example. Figure B.1 is equation (1.3)'s computational graph, but duplicated nodes are not merged. Notice that it is very different from Figure 1.2(a).

Cost Analysis

It is obvious that generation of unmerged graph takes $O(s)$ work where s is size of tape. To merge duplicated nodes is more expensive: every node is compared with other nodes with the same parent nodes, so running time is $O(M \cdot s)$ where M is maximum number of child nodes among all nodes. In practice, most nodes have only one or two child nodes, nodes merging performs like $O(s)$.

B.3 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 2 diving graph evenly in fact is in terms of work. Consider equation (1.7), 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.

References

[1] Thomas F. Coleman and Arun Verma, *The Efficient Computation of Sparsest Jacobian Matrices Using Automatic Differentiation.* SIAM J. SCI. COMPUT. Vol. 19, No. 4, pp. 1210-1233, July 1998.

[2] Rall, Louis B., *Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science.* 120. Springer. ISBN 0-540-10861-0, 1981.

[3] Christian H. Bischof, Peyvand M. Khademi, Ali Bouaricha, and Alan Carle, *Efficient Computation of Gradients and Jacobians by Transparent Exploitation of Sparsity in Automatic Differentiation.* 1996.

[4] Thomas F. Coleman and Gudbjorn F. Jonsson, *The Efficient Computation of Strucured Gradients Using Automatic Differentiation.* SIAM J. SCI. COMPUT. Vol. 20, No. 4, pp. 1430-1437, March 1999.

[5] Thomas F. Coleman and Wei Xu, *Fast (Structured) Newton Computations.* SIAM J. SCI. COMPUT. Vol. 31, No. 2, pp. 1175-1191, December 2008.

[6] L. R. Ford and D. R. Fulkerson, *Maximal Flow Through a Network.* Canadian Journal of Mathematics 8: 399-404, 1956.

[7] Sanjeev Arora, Elad Hazan, and Satyen Kale, $\sqrt{O(logn)}$ *Approximation to Sparsest Cut in* $\tilde{O}(n^2)$ *Time.* Annual symposium on foundations of computer science No45, Rome , ITALIE (17/10/2004), 2004.

[8] Bischof, Bouaricha, Khademi ,and Moré, *Computing Gradients in Large-Scale Optimization Using Automatic Differentiation.* INFORMS J. Computing, vol 9, 185-194, 1997.

[9] Coleman, Santosa, and Verma, *Efficient Calculation of Jacobian and Adjoint Vector Products in Wave Propagational Inverse Problems Using Automatic Differentiation.* Journal of Computational Physics 157, pp. 234-255, 2000.

[10] Coleman, Santosa, and Verma, *Semi-Automatic Differentiation, in Computational Methods for Optimal design and Control.* Jeff Borggaard, John Burns, Eugene Cliff, and Scott Schreck (eds), Birkhauser, pp. 113-126, 1998.

[11] Coleman and Verma, *Structure and Efficient Jacobian Calculation, in Computational Differentiation: Techniques, Applications, and Tools.* Martin Berz, Christian Bischof, George Corliss, Andreas Griewank (eds.), SIAM, Philadelphia, Penn., pp. 149-159, 1996.

[12] Averick, Moré, Bischof, Carle and Griewank, *Computing Large Sparse Jacobian matrices Using Automatic Differentiation.* SIAM Journal on Scientific Computing, 15, pp. 285 294, 1994.

[13] Bischof, Carle, Khademi, and Mauer, *ADIFOR2.0: Automatic Differentiation of Fortran 77 Programs.* IEEE Comput Sci. Eng., 3, 18-32, 1996.

[14] Bischof, Roh, and Mauer, *ADIC: An Extensible Automatic Differentiation Tool for ANSIO-C.* Software — Practice and Experience, 27, 1427-1456, 1997.

[15] Capriotti and Giles, *Algorithmic Differentiation: Adjoint Greeks Made Easy.* Computational Finance, 2011.

[16] Coleman and Verma, *Structure and Efficient Jacobian calculation.* Proceedings of the SIAM Workshop on Computational Differentiation, Sante Fe, February 1996.

[17] Coleman and Verma, *ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox.* ACM Transactions on Mathematical Software 22, pp. 150-175, 2000.

[18] Coleman and Cai, *The Cyclic Coloring Problem and Estimation of Sparse Hessian Matrices.* SIAM Journal on Algebraic and Discrete Methods 7, 221-235, 1986.

[19] Coleman and Verma, *Structure and Efficient Hessian Calculation.* in Advances in Nonlinear Programming, Proceedings of the 1996 International Conference on Nonlinear Programming, Ya-Xiang Yuan editor, pp. 57-72, Kluwer Academic Publishers, 1998.

[20] Griewank, Walther, *Evaluating Derivatives : Principles, and Techniques of Algorithmic Differentiation.* (2^{nd} ed.). SIAM, 2008.

[21] Griewank, *Direct Calculation of Newton Steps without Accumulating Jacobians, in Large-Scale Numerical Optimization.* T.F. Coleman and Y. Li, eds, SIAM, Philadelphia, Penn, pp115-137, 1990.

[22] Griewank and Corliss, *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* SIAM, Philadelphia, 1991.

[23] Griewank, Juedes, and Utke, *ADOL-C. A Package for The Automatic Differentiation of Algorithms Written in C/C++*, ACM Trans Math Software, 22, 131-167, 1996.

[24] Grebremedhin, Tarafdar, Manne, and Pothen, *New Acyclic and Star Coloring Algorithms with Applications to Computing Hessians.* SIAM J. Sci. Computing, 29, 1042-1072, 2007.

[25] Grebremedhin, Manne, and Pothen, *What Color is Your Jacobian? Graph Coloring for Computing Derivatives.* SIAM Review 47, pp 629-705, 2005.

[26] Hossian and Steihaug, *Computing a Sparse Jacobian Matrix by Rows and Columns.* Optim. Methods Software, 10, 33-48, 1998.

[27] Hossian and Steihaug, *Sparsity Issues in Computation of Jacobian Matrices.* Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, 123-130, 2002.

[28] Keyes, Hovland, McInnes, and Samyono, *Using Automatic Differentiation for Second-order Matrix-free Methods in PDE-constrained Optimization in Automatic Differentiation of Algorithms: From Simulatation to Optimization.* Springer-Verlag, 35-50, 2001.