Optimization for Derivatives, Derivatives for Optimization

13th INFORMS Computing Society Conference
January 6-8, 2013, Santa Fe, New Mexico

Paul D. Hovland, Todd S. Munson, Jean Utke
Argonne National Laboratory

Jieqiu Chen
Dow AgroSciences

Robert Luce
TU Berlin

The Automatic Differentiation Team @ Argonne

- Paul Hovland
- Boyana Norris
- Jean Utke
- Sri Hari Krishna Narayanan
- Azamat Mametjanov

- Alumni: J. Abate, C. Bischof, S. Bhowmick, **H. Cole-Mullen**, A. Griewank, P. Khademi, J. Kim, P. Malusare, U. Naumann, L. Roh, **I. Safro**, J. Shin, M. Strout, B. Winnicka

- Collaborators: P. Heimbach, L. Hascoët
- Funding: US DOE, NSF, NASA

Primary Messages

- Automatic differentiation is a good way to compute derivatives
  - Very efficient for gradients and directional derivatives
  - Very efficient for sparse and/or low-rank derivatives
  - Can be challenging to implement for general-purpose languages
- Efficient derivative computation relies on (approximate) solution to combinatorial optimization problems
  - Several heuristics exist
  - Heuristics are optimal or nearly so for certain special cases
  - In general, we do not know how good the heuristics are
  - Interested in IP formulations to evaluate effectiveness of heuristics, provide optimal solutions for important (sub-) problems
- Very interested in collaborations and/or hosting students/visitors

Outline

- Introduction to automatic differentiation (AD)
- Some (optimization) applications of AD
- Derivative computation as a graph problem (vertex elimination)
- Integer programming formulation of the vertex elimination problem

AD in a Nutshell

- Technique for computing analytic derivatives of programs
- Derivatives used in optimization, nonlinear PDEs, sensitivity analysis, inverse problems, uncertainty quantification, etc.
- AD = analytic differentiation of elementary functions + propagation by chain rule
  - Every programming language provides a limited number of elementary math functions
  - Thus, every function computed by a program may be viewed as the composition of these so-called intrinsic functions
  - Derivatives for the intrinsic functions are known and can be combined using the chain rule of differential calculus
- Less work than hand coding, more accurate than finite diffs
- Chain rule associativity leads to two main modes: forward and reverse
- Implemented using source transformation or operator overloading

Example: a simple function

```
#include <math.h>
#include <stream.h>

void func(double *f, double x, double y){
  double a,b;

  if (x > y) {
    a = cos(x);
    b = sin(y)*y*y;
  } else {
    a = x*sin(x)/y;
    b = exp(y);
  }
  *f = exp(a*b);
}
```

Example: AD via operator overloading

```
#include <math.h>
#include <stream.h>
#include "adouble.hxx"

void func(a_double *f, a_double x, a_double y){
  a_double a,b;

  if (x > y) {
    a = cos(x);
    b = sin(y)*y*y;
  } else {
    a = x*sin(x)/y;
    b = exp(y);
  }
  *f = exp(a*b);
}
```

Example: ADIC output

```
#include "ad_deriv.h"
#include <math.h>
#include "adintrinsics.h"
void   ad_func(DERIV_TYPE  *f,DERIV_TYPE  x,DERIV_TYPE  y) {
DERIV_TYPE  a, b, ad_var_0, ad_var_1, ad_var_2;
double  ad_adji_0,ad_loc_0,ad_loc_1,ad_adj_0,ad_adj_1,ad_adj_2,ad_adj_3;

   if (DERIV_val(x) > DERIV_val(y))     {
     DERIV_val(a) = cos( DERIV_val(x)); /*cos*/
      ad_adji_0 = -sin( DERIV_val(x));
        {
            ad_grad_axpy_1(&(a), ad_adji_0, &(x));
        }
     DERIV_val(ad_var_0) = sin( DERIV_val(y)); /*sin*/
      ad_adji_0 = cos( DERIV_val(y));
        {
            ad_grad_axpy_1(&(ad_var_0), ad_adji_0, &(y));
        }
        {
            ad_loc_0 = DERIV_val(ad_var_0) * DERIV_val(y);
            ad_loc_1 = ad_loc_0 * DERIV_val(y);
            ad_adj_0 = DERIV_val(ad_var_0) * DERIV_val(y);
            ad_adj_1 = DERIV_val(y) * DERIV_val(y);
            ad_grad_axpy_3(&(b), ad_adj_1, &(ad_var_0), ad_adj_0, &(y), ad_loc_0, &(y));
            DERIV_val(b) = ad_loc_1;
        }
    }
    else     {
        // ...
```

Capabilities

- Fast (O(1) function evaluation) computation of
  - gradient (reverse)
  - Jacobian-vector product Jv (forward)
  - transposed-Jacobian-vector product $J^T w$ (reverse)
  - Hessian-vector product (F+R, R+F, R+R)
- Efficient computation of full Jacobians and Hessians, when able to exploit (approximate) sparsity and/or low-rank structure
- Efficient high-order (directional) derivatives for modest number of independent variables / directions

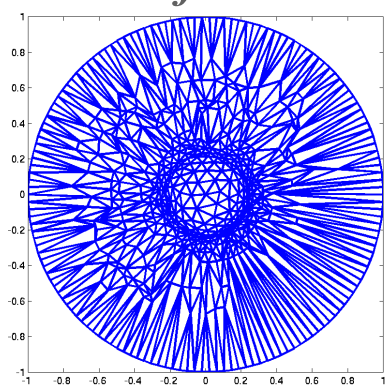
Challenges

- Full Jacobian or Hessian with no underlying structure (rare) can be expensive to compute
- Reverse mode requires storage of intermediate states
  - Worst case storage proportional to # flops in function evaluation
  - In practice, use combination of storage/recomputation to reduce storage
- Implementation of source transformation tools requires robust compiler infrastructure and compiler analyses
- Computes (chain-rule) derivatives of code

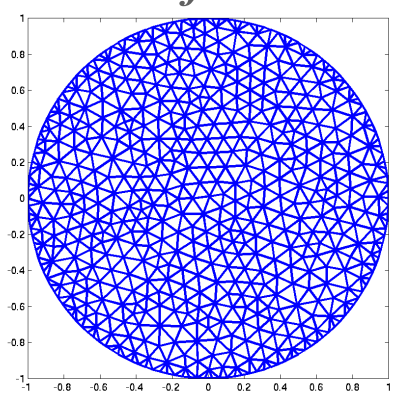
Application: mesh quality optimization

- Optimization used to move mesh vertices to create elements as close to equilateral triangles/tetrahedrons as possible
- Semi-automatic differentiation is 10-25% faster than hand-coding for gradient and 5-10% faster than hand-coding for Hessian
- Automatic differentiation is a factor 2-5 times faster than finite differences

Before

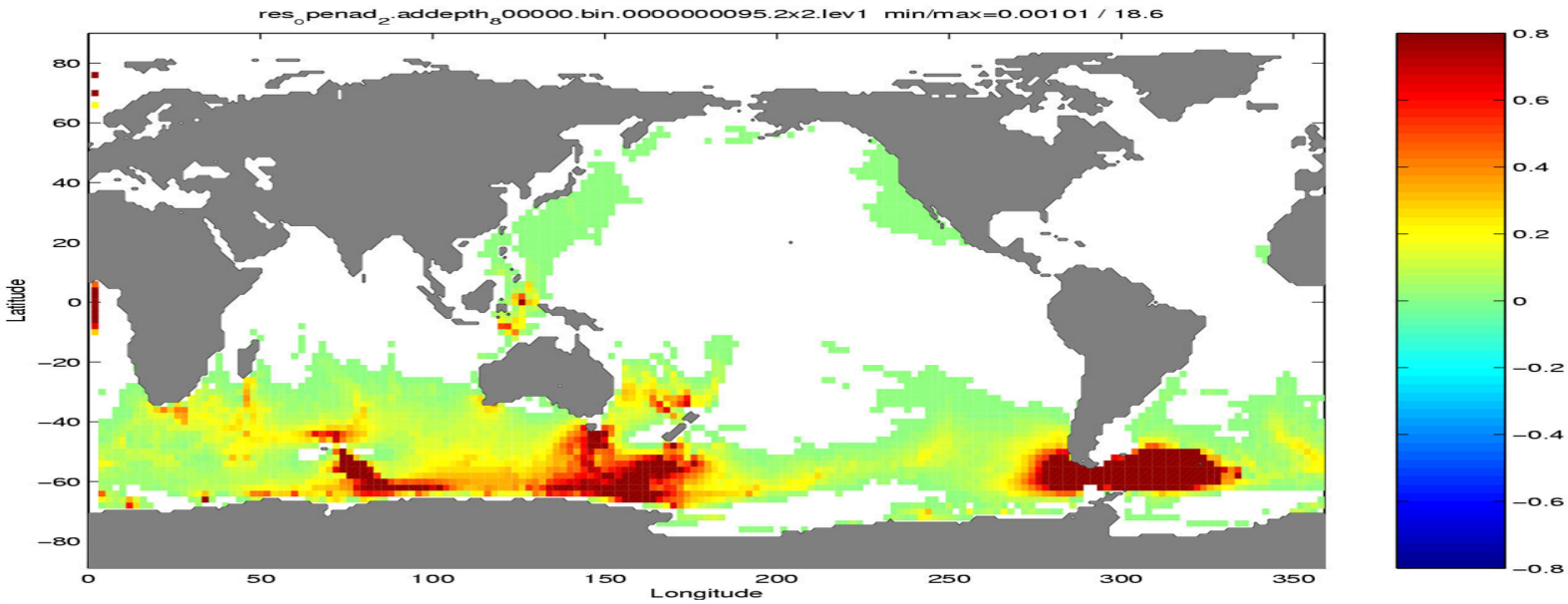


After



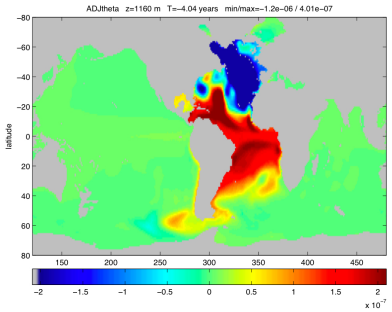
Application: Sensitivity analysis in simplified climate model

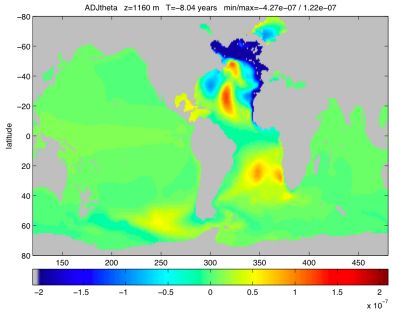
- Sensitivity of flow through Drake Passage to ocean bottom topography
  - Finite difference approximations: 23 days
  - Naïve automatic differentiation: 2 hours 23 minutes
  - Smart automatic differentiation: 22 minutes

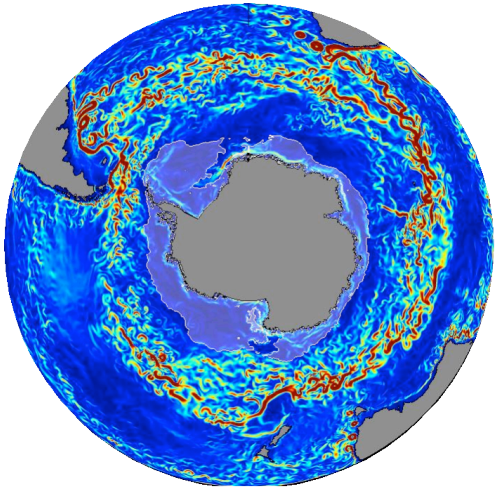


Application: Sensitivities for State Estimation

- Facilitated through the use of reverse mode automatic differentiation tools
- Provide the full gradient vector: first derivatives with respect to potentially millions or billions of independent variables
- Moderate cost : a small multiple of the forward model alone), independent of the number of independent variables
- Example: MITgcm
  - One simulation run (20 yrs at 4º): 52 cpu-hours
  - Gradient using AD: 204 hrs (8.5 cpu-days)
  - Finite-difference gradient approximation: 1.1 million cpu-years
  - Goal: O(10)-O(100) gradient evaluations at 1/2º



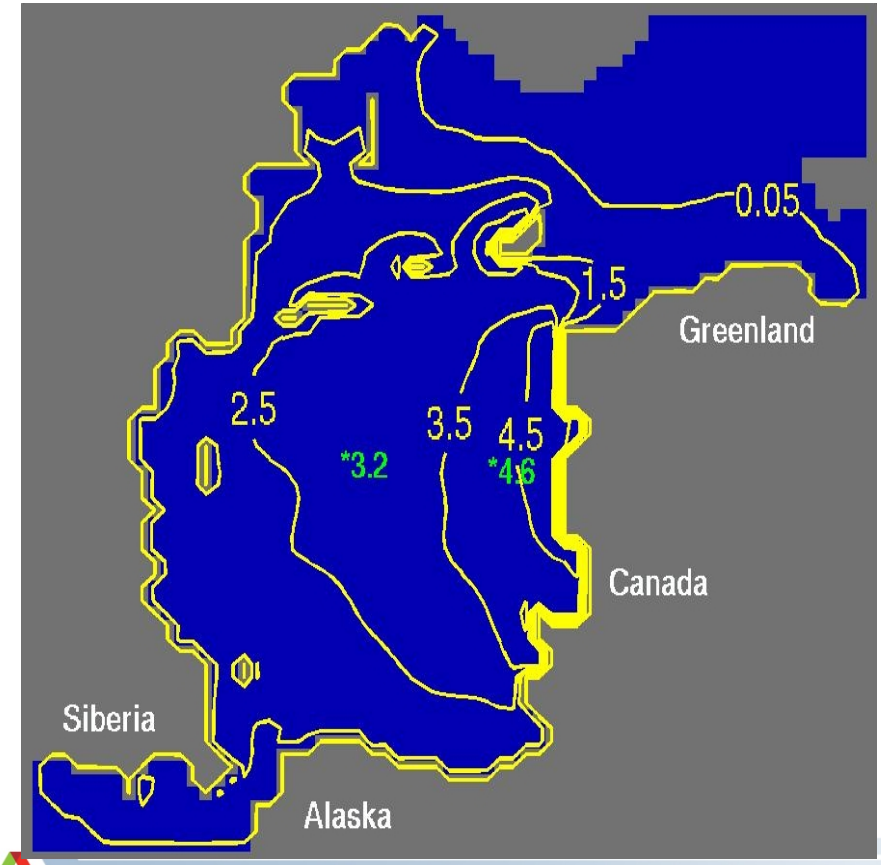




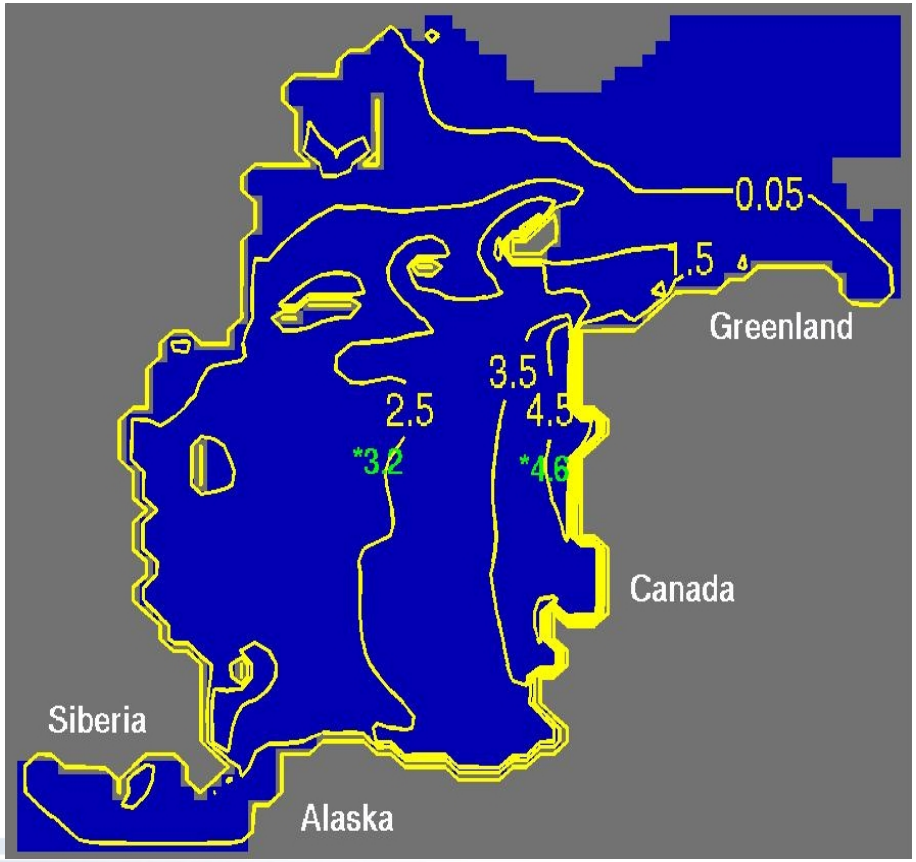
Parameter estimation: sea ice model

- *Simulated (yellow) and observed (green) March ice thickness (m)*

Tuned parameters



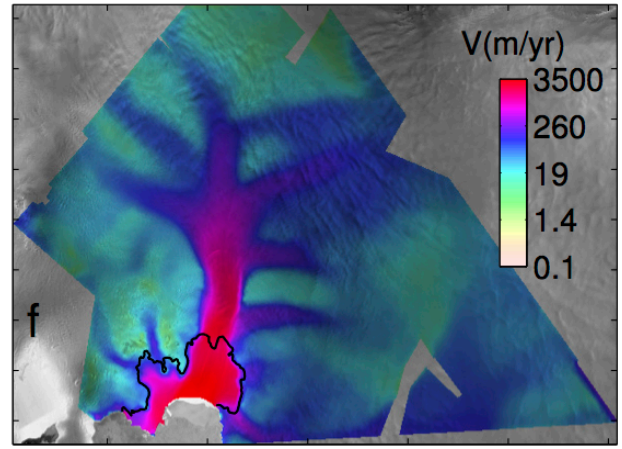
Standard parameters

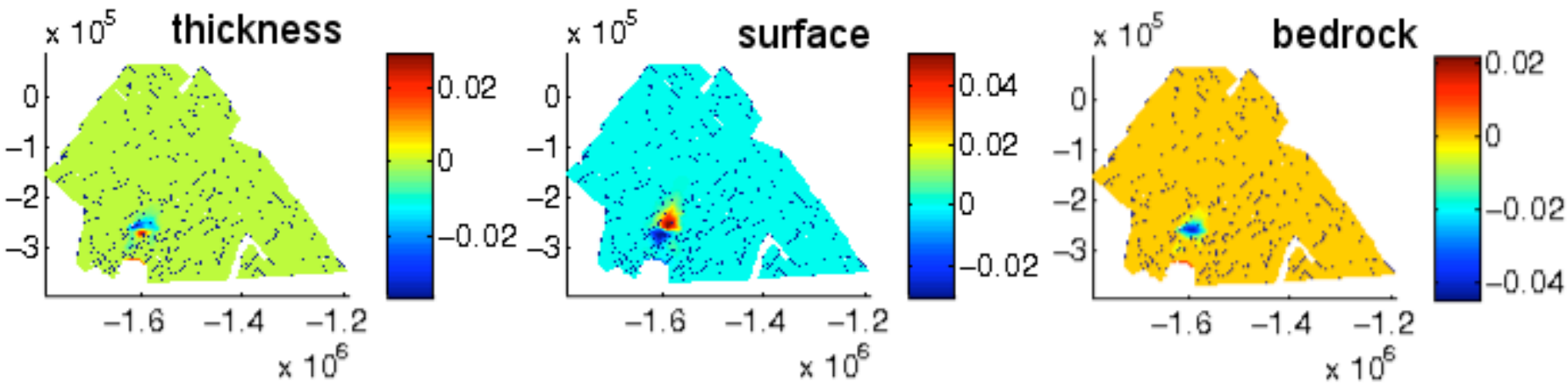


Ice Sheet Grounding Line Analysis

- grounding line = ice sheet starts to float
- question of interest: sensitivities of ice mass flux at the grounding line
- has impact on sea level rise

Pine Island Glacier



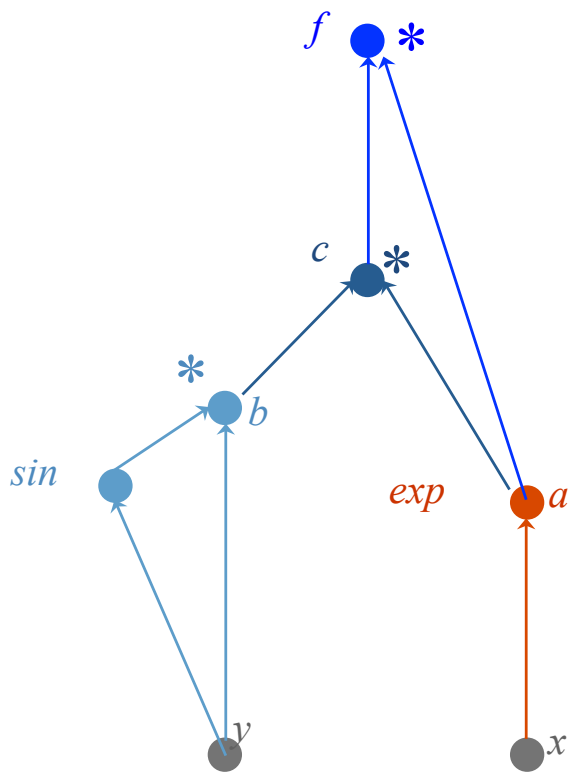


Optimal Derivative Accumulation

- Represent function using a directed acyclic graph (DAG)
- Computational graph
  - Vertices are intermediate variables, annotated with function/operator
  - Edges are unweighted
- Linearized computational graph
  - Edge weights are partial derivatives
  - Vertex labels are not needed
- Compute sum of weights over all paths from independent to dependent variable(s), where the path weight is the product of the weights of all edges along the path [Baur & Strassen]
- Find an order in which to compute path weights that minimizes cost (flops): identify common subpaths (=common subexpressions in Jacobian)

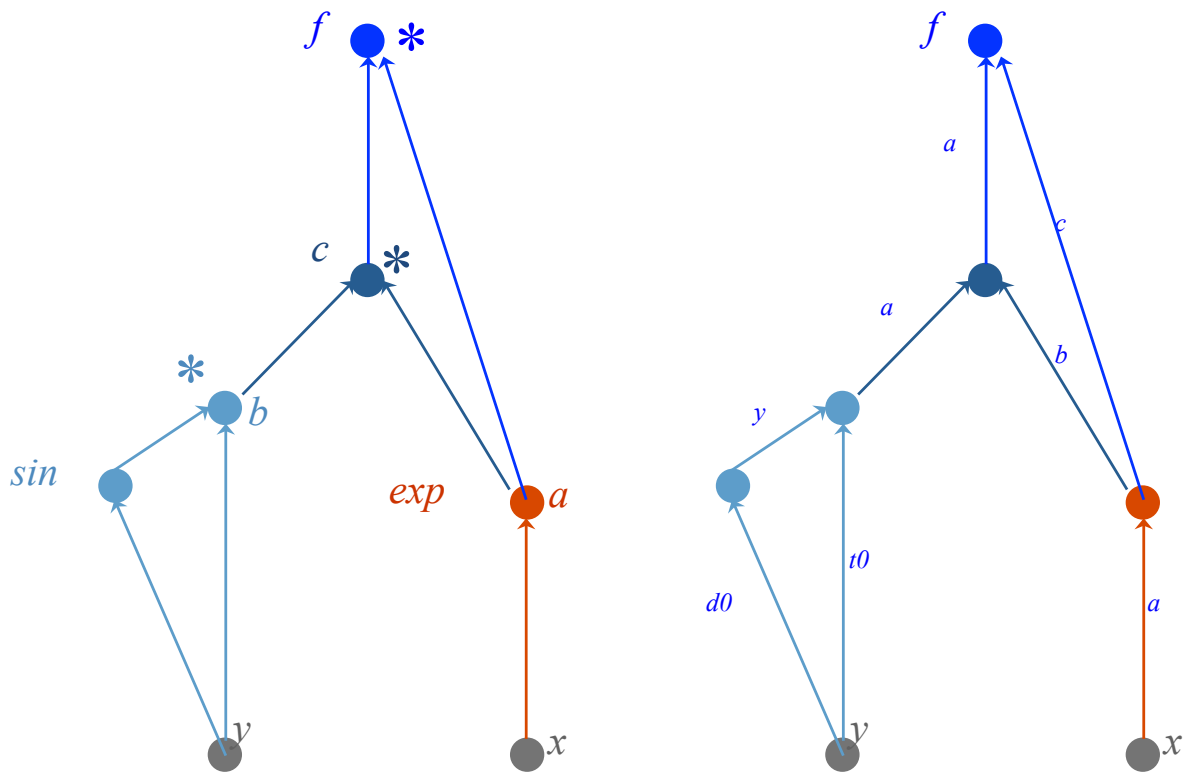
A simple example

$b = sin(y)*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

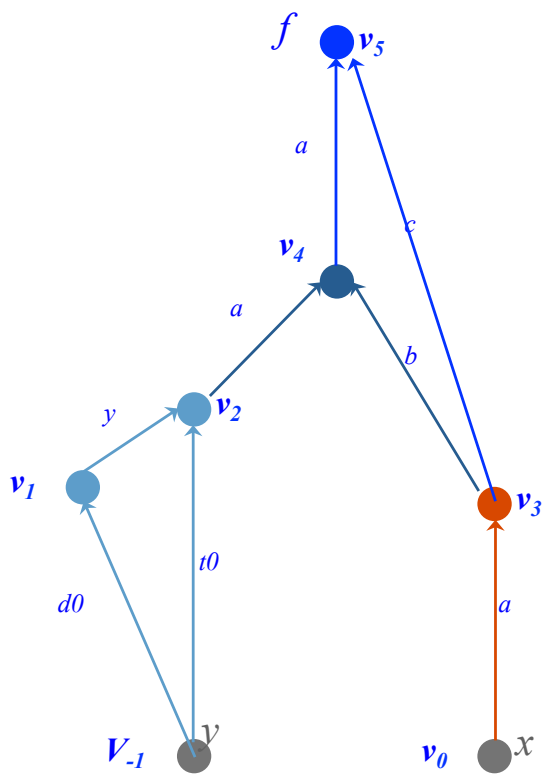


A simple example

$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$



Brute force

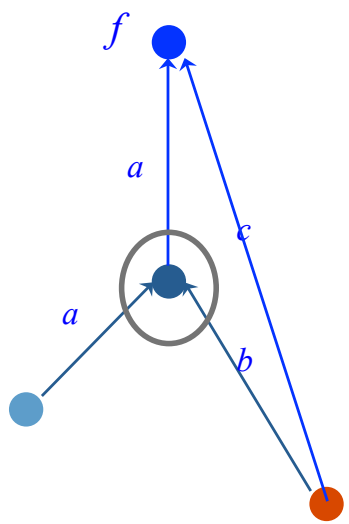


- Compute products of edge weights along all paths
- Sum all paths from same source to same target
- Hope the compiler does a good job recognizing common subexpressions

$$dfdy = d0*y*a*a + t0*a*a$$
$$dfdx = a*b*a + a*c$$

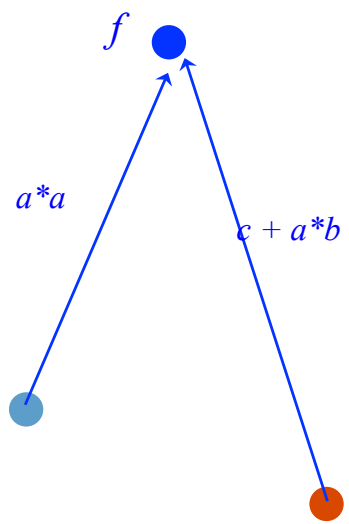
8 mults 2 adds

Vertex elimination



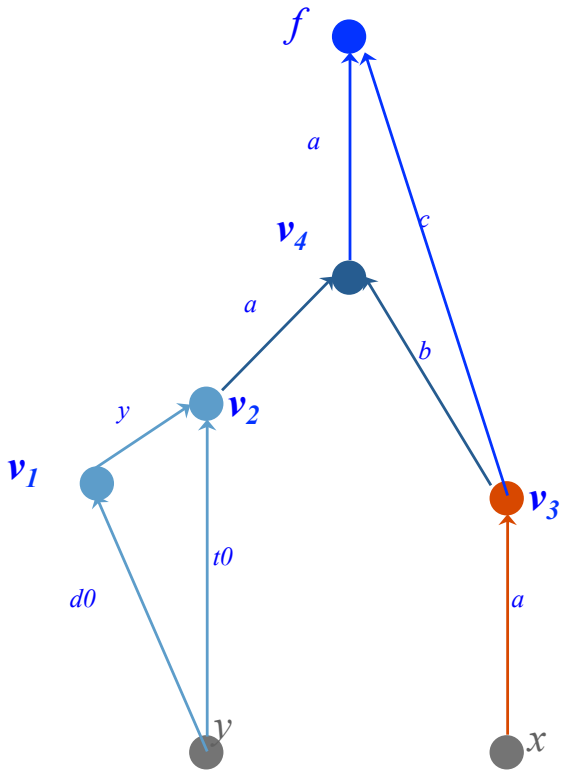
- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph
- Cost is the in degree times the out degree

Vertex elimination



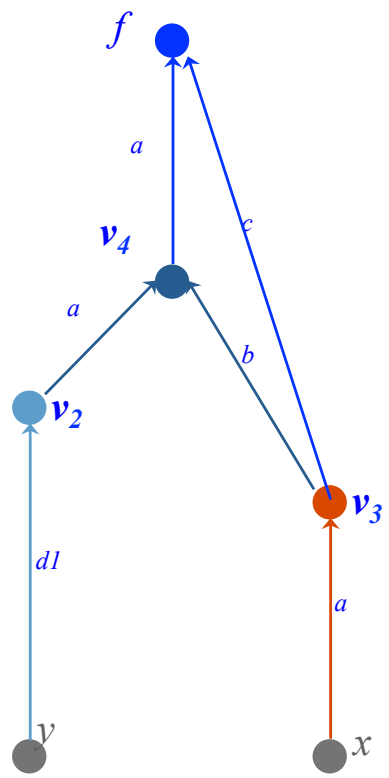
- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph
- Cost is the in degree times the out degree

Forward mode: eliminate vertices in topological order



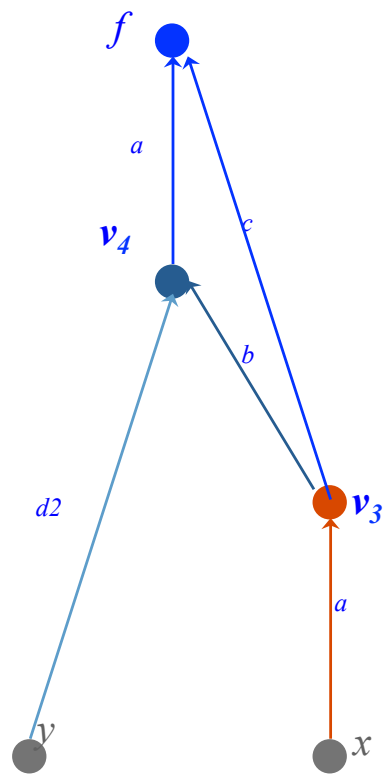
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

Forward mode: eliminate vertices in topological order



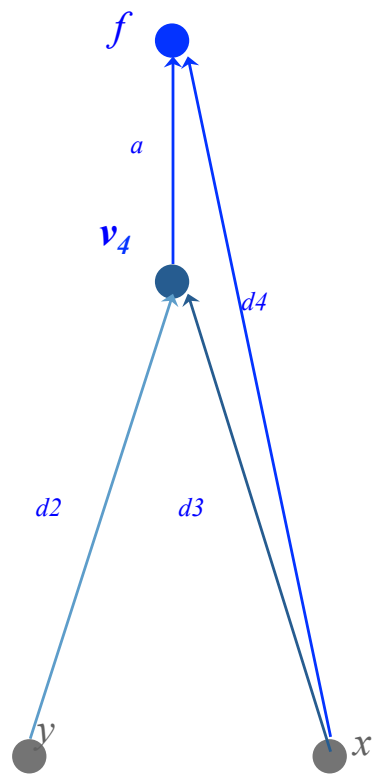
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$

Forward mode: eliminate vertices in topological order



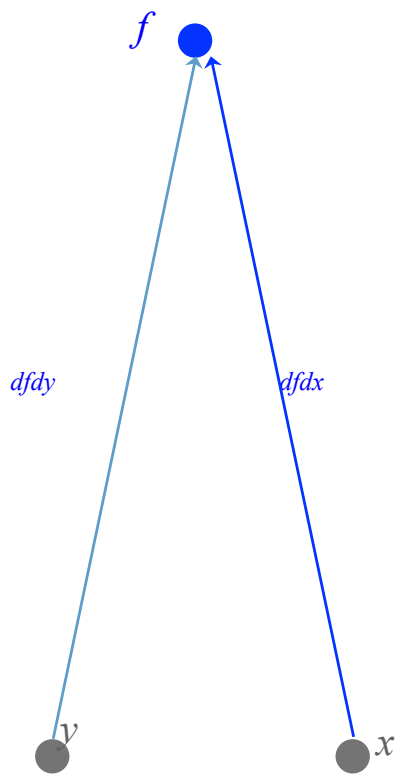
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$

Forward mode: eliminate vertices in topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$
$d3 = a*b$
$d4 = a*c$

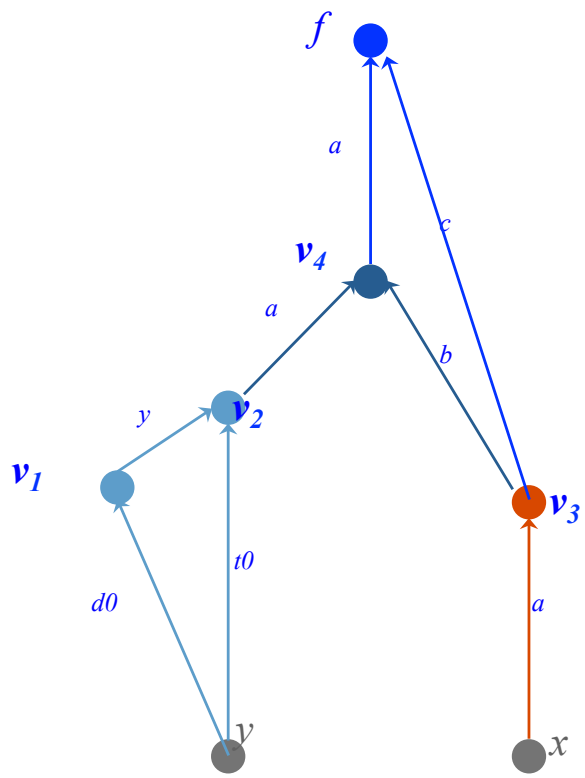
Forward mode: eliminate vertices in topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$
$d3 = a*b$
$d4 = a*c$
$dfdy = d2*a$
$dfdx = d4 + d3*a$

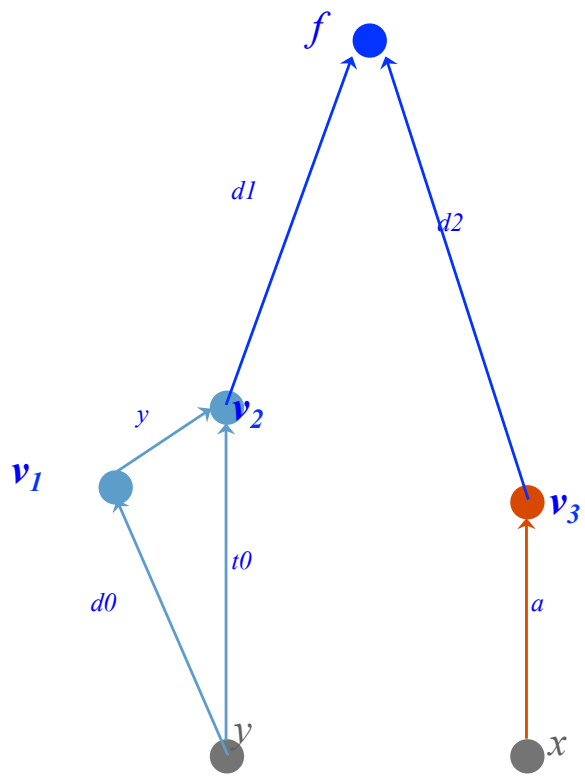
6 mults 2 adds

Reverse mode: eliminate in reverse topological order



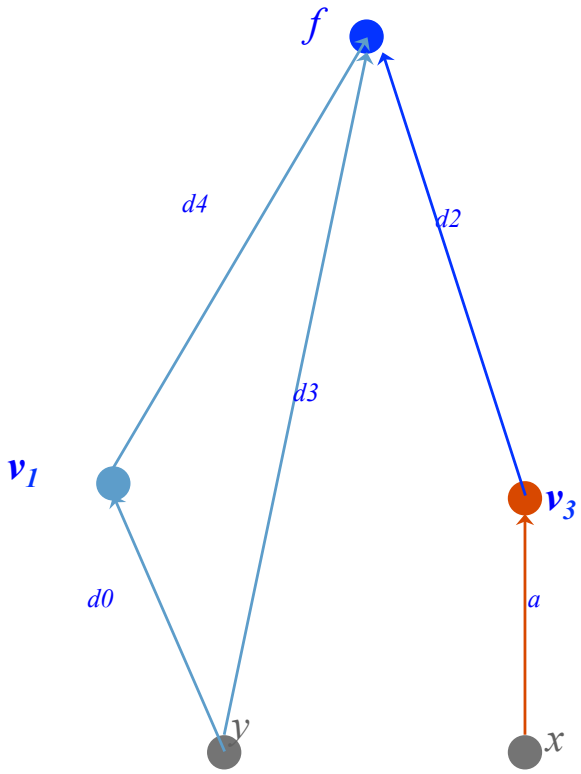
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

Reverse mode: eliminate in reverse topological order



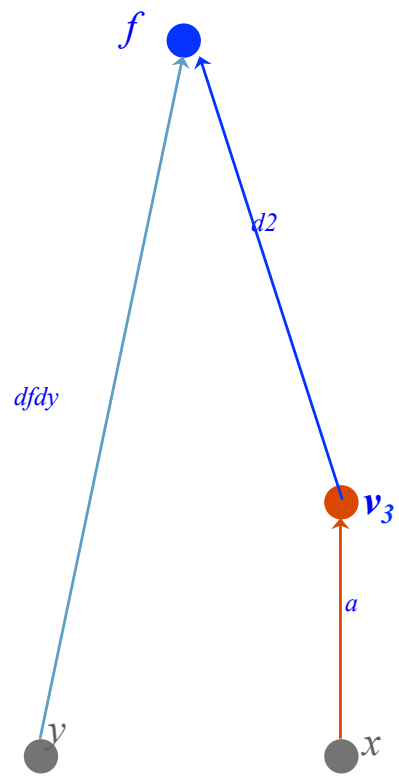
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$

Reverse mode: eliminate in reverse topological order



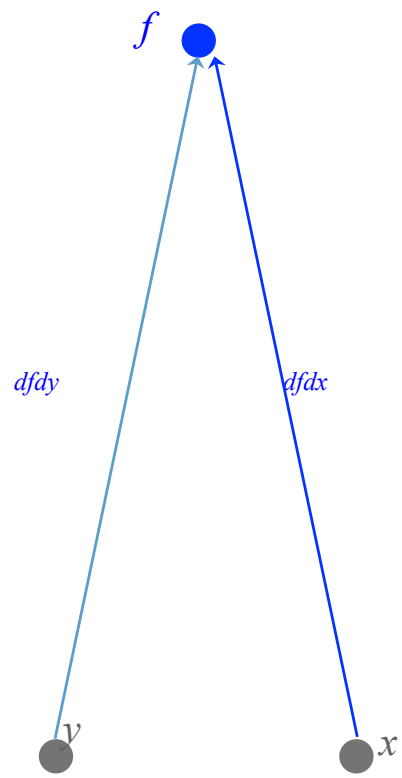
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$

Reverse mode: eliminate in reverse topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$
$dfdy = d3 + d0*d4$

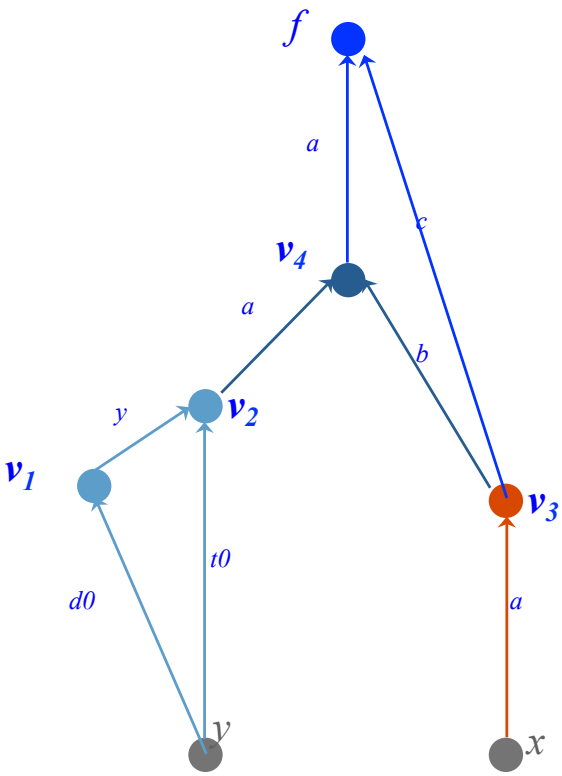
Reverse mode: eliminate in reverse topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$
$dfdy = d3 + d0*d4$
$dfdx = a*d2$

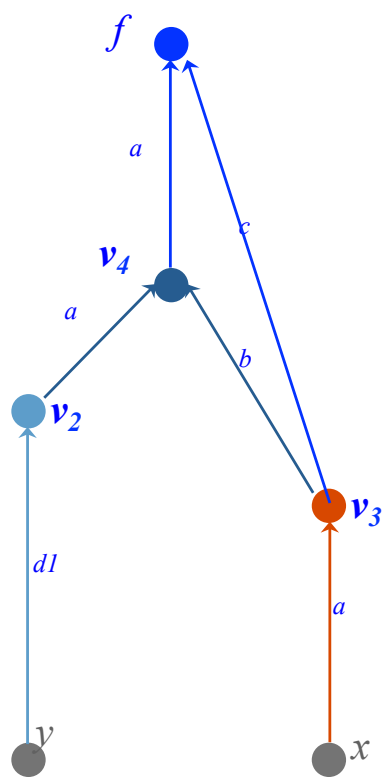
6 mults 2 adds

“Cross-country” mode



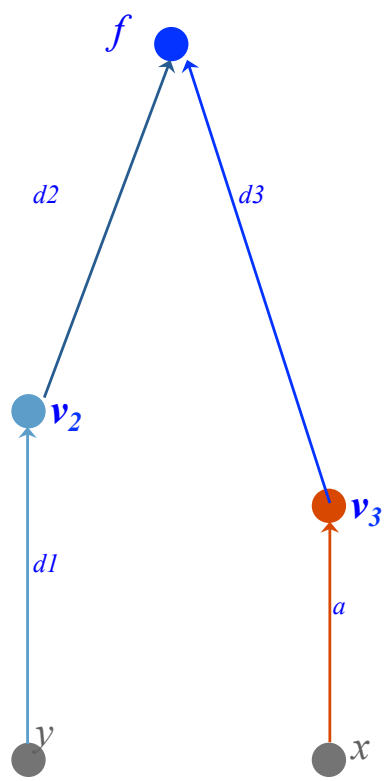
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

“Cross-country” mode



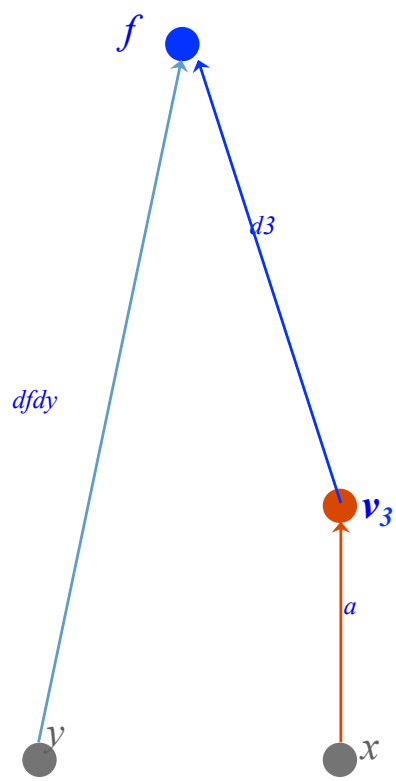
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$

“Cross-country” mode



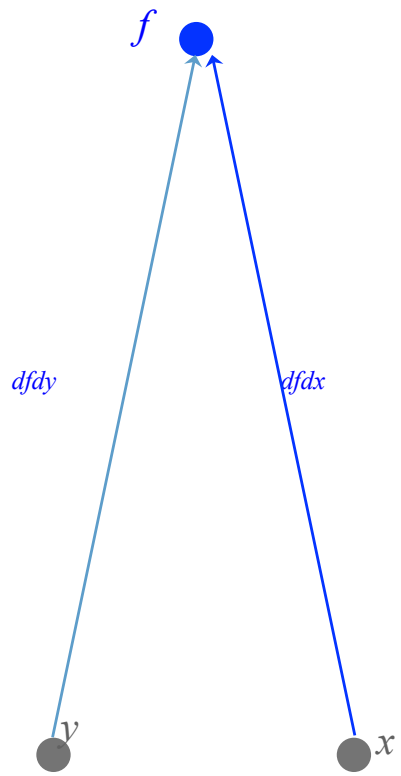
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$

“Cross-country” mode



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$
$dfdy = d1*d2$

“Cross-country” mode



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$
$dfdy = d1*d2$
$dfdx = a*d3$

5 mults 2 adds

Optimal vertex elimination order

- Goal: use integer programming to find the optimal vertex elimination order
- Primary motivation: evaluate effectiveness of heuristics
- Secondary motivation: find optimal elimination order for key computational kernels

Approach

- Model graph at each elimination step as a matrix
- Require consistency between steps k and k+1 (but do not enforce sequence)
  - flops_k+1 = flops_k + in(x_k)*out(x_k)
  - Matrix C_k+1 = C_k with row/col x_k removed, appropriate edges added
- Provide problem-specific lower bounds to assist IP solver
  - Current lower bound is:

$$flops \geq flops_k + l(x_v) + \sum_{x \notin x_1 \ldots x_k, x \neq x_v} \rho(x)$$

    where l(x_v) is the cost of eliminating vertex x_v last and ρ(x) is the size of the minimal vertex separator between the independents and x times the size of the minimal vertex separator between x and the dependents
  - Requires solving auxiliary problems to determine sizes of vertex separators

Integer Programming Formulation

$$\text{minimize} \quad F := \sum_{(i,j)\in\tilde{E}} \sum_{t\in T} f_{ijt}$$

Minimize total # multiplications

$$\text{subject to} \quad \sum_{i\in N} x_{it} = 1$$

Remove one vertex a time

$$\sum_{t\in T} x_{it} = 1$$

Each vertex must be removed

$$\left.\begin{aligned} e_{ijt} &\geq x_{it} + c_{ijt} - 1 \\ e_{ijt} &\geq x_{jt} + c_{ijt} - 1 \\ e_{ijt} &\leq x_{it} + x_{jt} \\ e_{ijt} &\leq c_{ijt} \end{aligned}\right\}$$

Edges to be eliminated

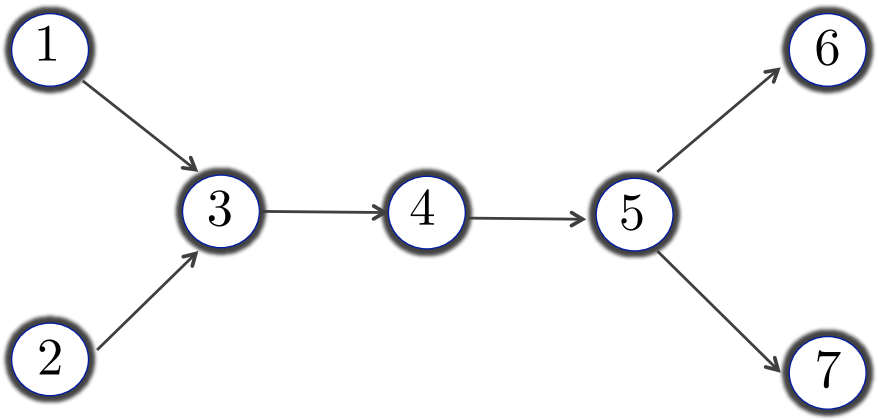
$$f_{ijt} \geq e_{ikt} + e_{kjt} - 1$$

Edges generated

$$\left.\begin{aligned} c_{ij(t+1)} &\geq f_{ijt} \\ c_{ij(t+1)} &\leq 1 - e_{ijt} \\ c_{ij(t+1)} &\leq c_{ijt} + (f_{ijt} + e_{ijt}) \\ c_{ij(t+1)} &\geq c_{ijt} - (f_{ijt} + e_{ijt}) \end{aligned}\right\}$$

Update adjacency matrix

Symmetry-breaking Constraints



Observation:
If ***i*** and ***j*** are NOT adjacent, then sequence $(\ldots i, j, \ldots)$ and $(\ldots, j, i, \ldots)$ have same cost.

$$COST(3, 5, 4) = COST(5, 3, 4)$$

=1, if remove *j* at time t

=1, if remove *i* at time t+1

=1, if not adjacent

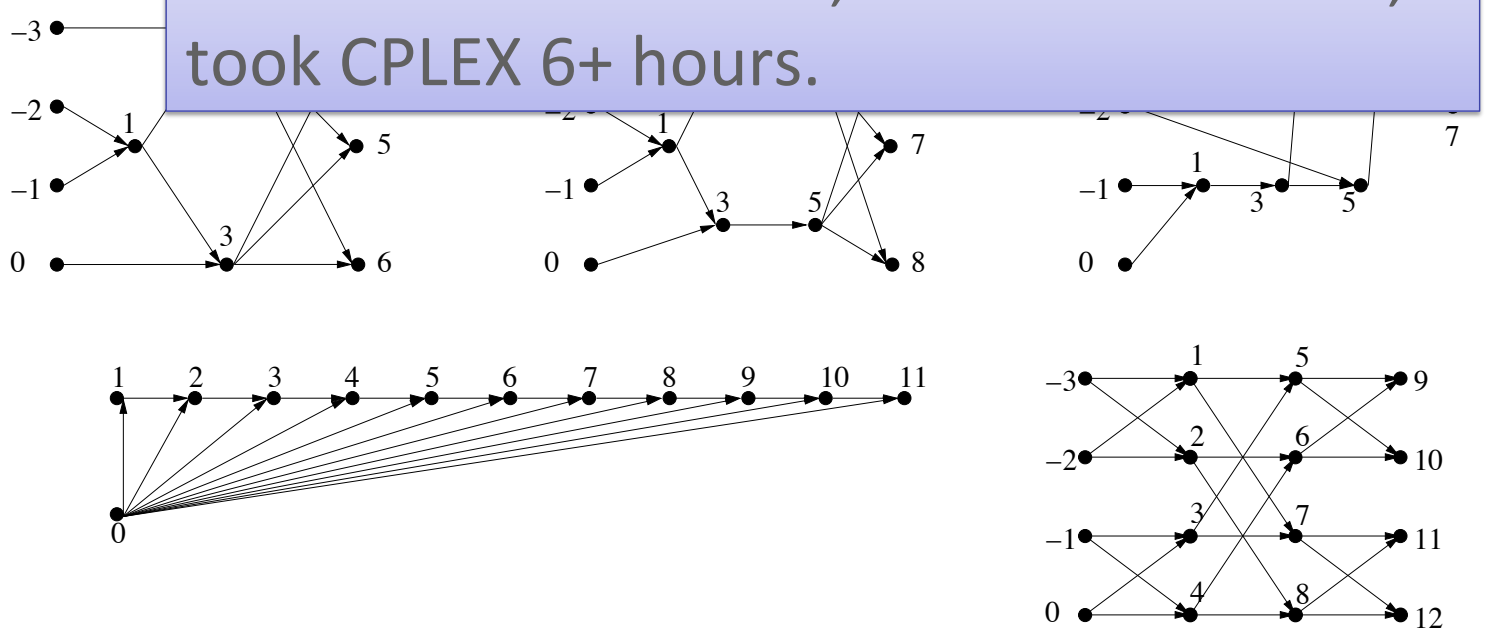
$$x_{jt} + x_{i(t+1)} + (1 - c_{ijt}) \leq 2$$

Sequence $(\ldots, j, i, \ldots)$ is not allowed!

Experimental Results

Problem	Vertices			Number mults				Times			LP bounds of M1
	|X|	|Y|	|Z|	For	Rev	Mark	Opt	M0	M1	M2	
Fig 10.4	4	3	3	22	18	22	18	0.04	0.02	0.11	18
Ex 10.8	4	3	5	28	24	26	22	0.26	0.37	0.77	16
Fig 10.1	4	2	5	20	18	16	15	0.71	0.35	0.29	13
Revbound											
Butterfly											

Largest problem solved: 100 intermediate vertices, 131 total vertices, took CPLEX 6+ hours.



Other things we tried

- low_bnd_0: flops ≥ v
- low_bnd_1: flops ≥ flops_fwd/2 (if #indeps = 1)
- low_bnd_1: flops ≥ flops_rev/2 (if #deps = 1)
- low_bnd_2: flops ≥ flops_k + (v-k)
- low_bnd_2p: flops ≥ flops_k + $\sum$(v-k-1) + last(x_v)
  last(x_v) is the cost of eliminating x_v

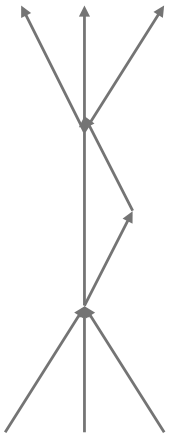
- We initially enforced a sequence, so that x_k+1 could not be chosen unless x_1 … x_k were also chosen.. This was a bad idea.

Practical Matters: constructing computational graphs

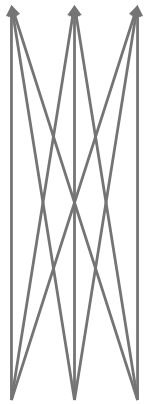
- At compile time (source transformation)
  - Structure of graph is known, but edge weights are not: in effect, implement inspector (symbolic) phase at compile time (offline), executor (numeric) phase at run time (online)
  - In order to assemble graph from individual statements, must be able to resolve aliases, be able to match variable definitions and uses
  - Scope of computational graph construction is usually limited to statements or basic blocks
  - Computational graph usually has O(10)—O(100) vertices
- At run time (operator overloading)
  - Structure and weights both discovered at runtime
  - Completely online—cannot afford polynomial time algorithms to analyze graph
  - Computational graph may have O(10,000) vertices

Minimal graph of a Jacobian (scarcity)

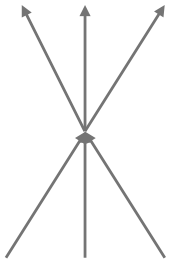
Original DAG



Bipartite DAG



Minimal DAG



Reduce graph to one with minimal number of edges (or smallest number of DOF)

How to find the minimal graph? Relationship to matrix properties?

Avoid "catastrophic fill in" (empirical evidence that this happens in practice)

In essence, represent Jacobian as sum/product of sparse/low-rank matrices

Conclusions & Future Work

- AD can provide derivatives for optimization
- IP model for Jacobian accumulation by vertex elimination
- Some success in computationally solving the IP
- Future directions
  - New applications, especially for directional derivatives
  - Develop IP models for other AD problems, e.g., scarcity
  - Integrate IP model into tools (primarily for validation)
  - Develop SDP/alternative relaxation
  - Benchmark for AD heuristics algorithms
- Students (and postdocs?) wanted: hovland@anl.gov or http://www.mcs.anl.gov/about/education.php

For More Information

- J Chen, P Hovland, T Munson, J Utke, "An Integer Programming Approach to Optimal Derivative Accumulation" in *Recent Advances in Algorithmic Differentiation*, Springer, 2012.
- Andreas Griewank, Evaluating Derivatives, SIAM, 2000.
- Griewank, "On Automatic Differentiation"; this and other technical reports available online at: http://www.mcs.anl.gov/autodiff/tech_reports.html
- AD in general: http://www.mcs.anl.gov/autodiff/, http://www.autodiff.org/
- ADIC: http://www.mcs.anl.gov/adic/
- OpenAD: http://www.mcs.anl.gov/openad/
- E-mail: hovland@mcs.anl.gov