2016 SIAM WORKSHOP ON COMBINATORIAL SCIENTIFIC COMPUTING

10 OCTOBER 2016

AN INTEGER PROGRAMMING FORMULATION OF THE MINIMAL JACOBIAN REPRESENTATION PROBLEM

PAUL HOVLAND
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA

Argonne NATIONAL LABORATORY

OUTLINE

- Introduction to Automatic/Algorithmic Differentiation (AD)
- A graph model of AD
- Preaccumulation and scarcity
- Experimental results
- Conclusions

AUTOMATIC/ALGORITHMIC DIFFERENTIATION

AD in a Nutshell

- Technique for computing analytic derivatives of functions computed by programs (potentially millions of lines of code)
- 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 mathematical 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

AUTOMATIC/ALGORITHMIC DIFFERENTIATION

Forward Mode AD

- Start with independent variables and follow flow of the original function computation
- Computes Jacobian times a matrix S
- Cost is proportional to the number of columns in S
- Special case: Jv costs a small constant times the cost of the function
- Ideal for functions with a small number of independent variables
- Partial derivatives associated with intermediate variables are used at the same time as the variables themselves

AUTOMATIC/ALGORITHMIC DIFFERENTIATION

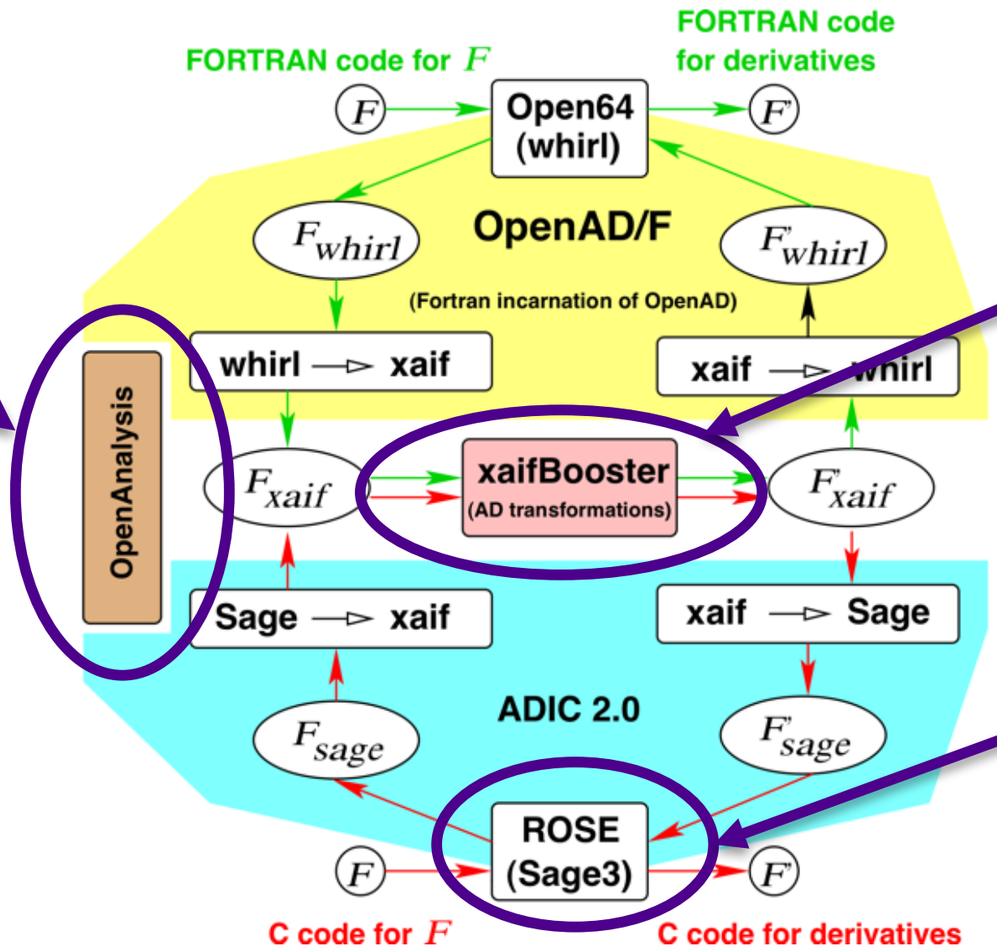
Reverse/Adjoint Mode AD

- Start with dependent variables and propagate derivatives back to independent variables
- Computes matrix W times Jacobian
- Cost is proportional to the number of rows in W
- Special case: $J^T v$ costs a small constant times the cost of the function
- Ideal for functions with a small number of dependent variables
- Intermediate partial derivatives must be stored or recomputed – in the worst case storage grows with the number of operations in the function
- Control flow must be reversed (must store or reproduce control flow decisions)

AUTOMATIC/ALGORITHMIC DIFFERENTIATION

AD Tool Implementation

Domain-specific data-flow analyses



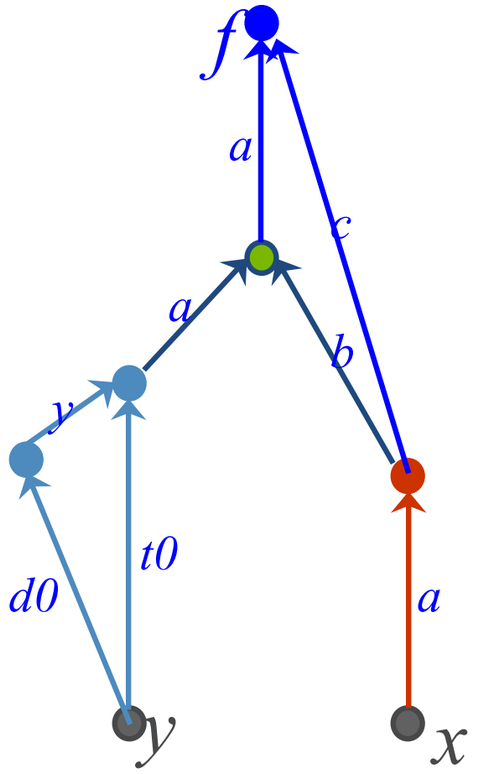
Combinatorial algorithms

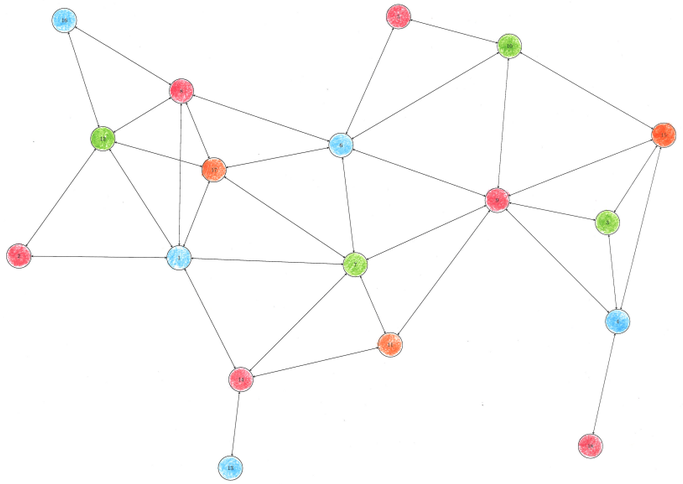
Parse/unparse

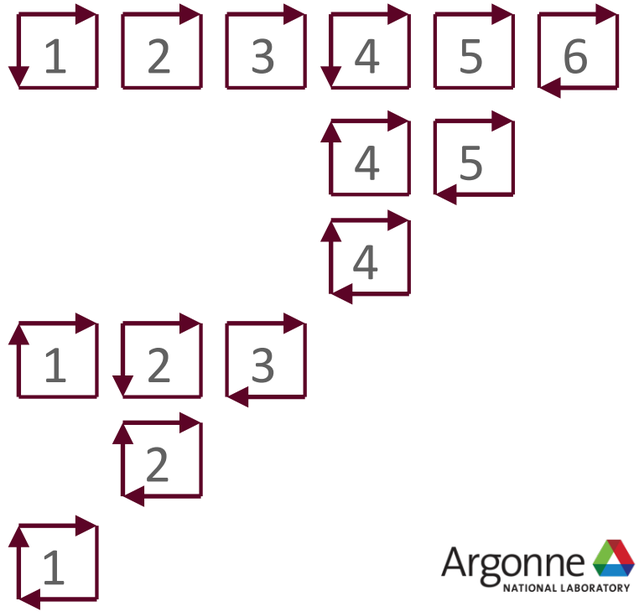
AUTOMATIC/ALGORITHMIC DIFFERENTIATION

Combinatorial problems in AD

- Minimize ops to compute Jacobian
  - Exploit chain rule associativity
  - Related to min fill in factorization
- Minimal representation
  - Minimize edge count in DAG
  - Jacobian as the sum/product of sparse/low-rank matrices
- Adjoint recompute/store tradeoff
  - G&W: Minimize recomputation
  - Aupy et al.: minimize time
- Matrix (graph) coloring
  - Minimize columns in JS







A GRAPH MODEL OF AD

Accumulating derivatives

- 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 GRAPH MODEL OF AD

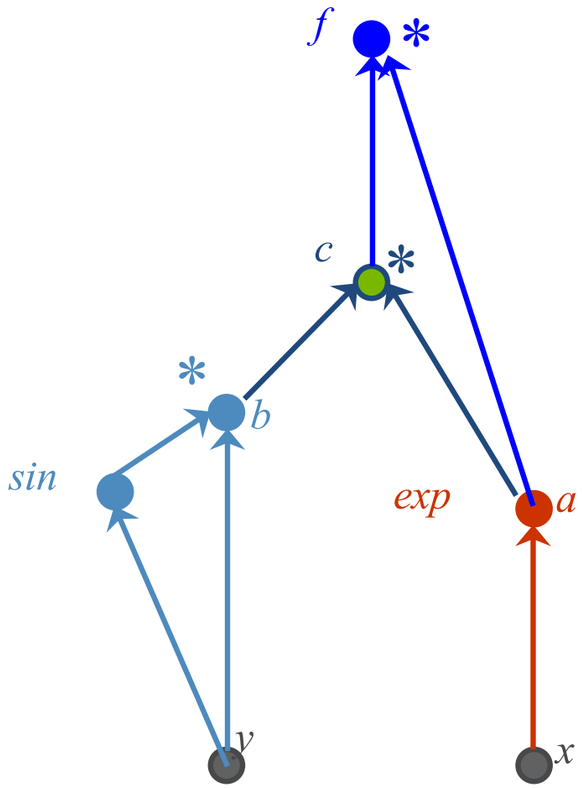
A simple example

$b = sin(y) * y$

$a = exp(x)$

$c = a * b$

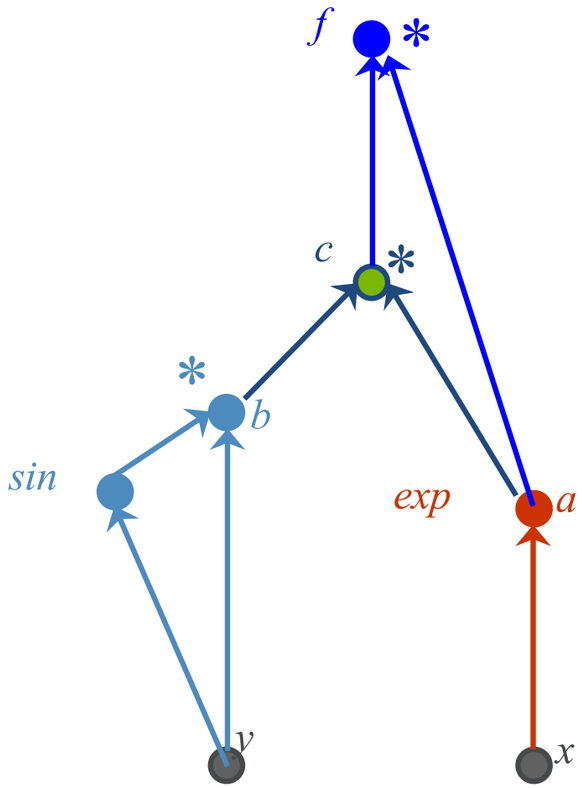
$f = a * c$

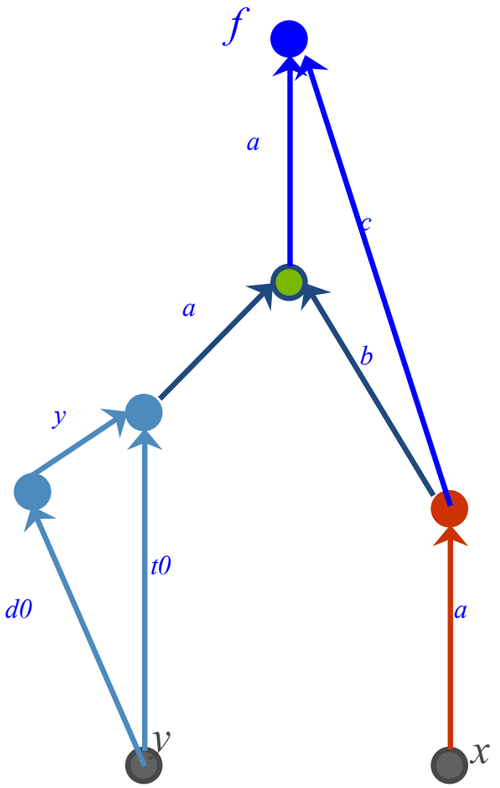


A GRAPH MODEL OF AD

A simple example

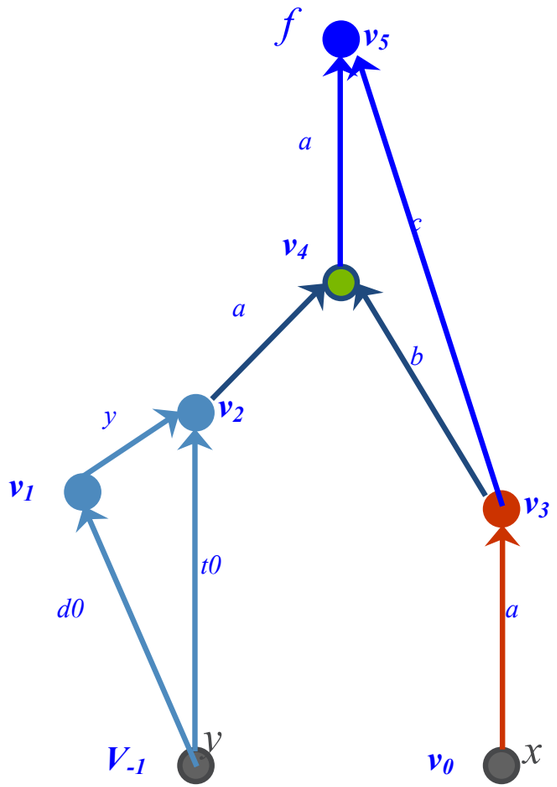
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$





A GRAPH MODEL OF AD

Brute force



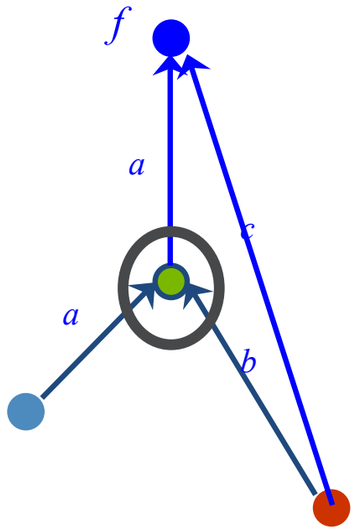
- Compute products of edge weights along all paths
- Sum all paths from same source to same target
- Hope the compiler does a good job recognizing common subexpressions

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

8 mults 2 adds

A GRAPH MODEL OF AD

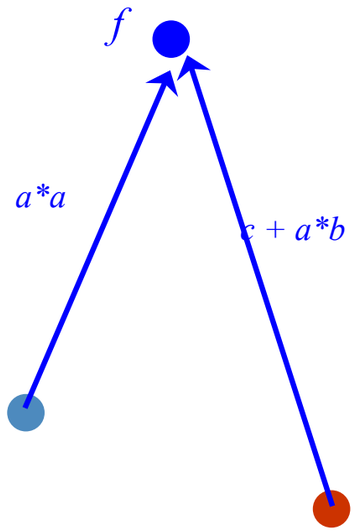
Vertex elimination



- 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

A GRAPH MODEL OF AD

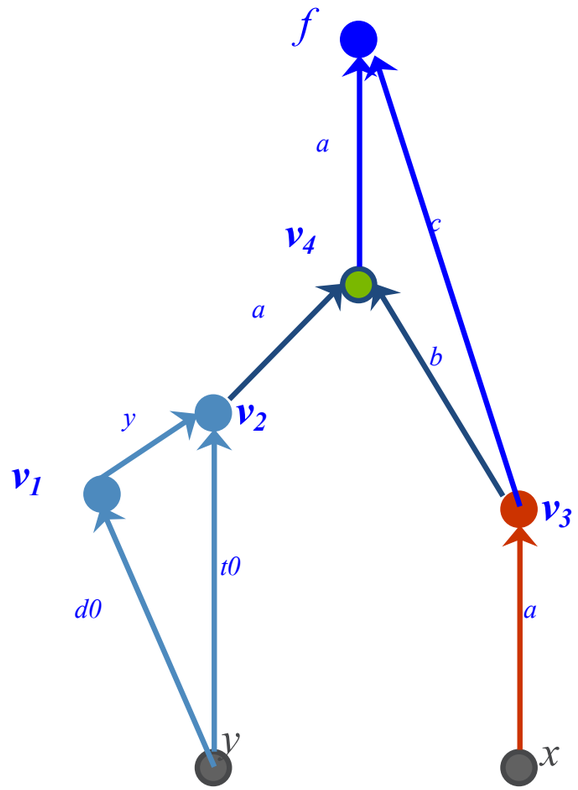
Vertex elimination



- 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

A GRAPH MODEL OF AD

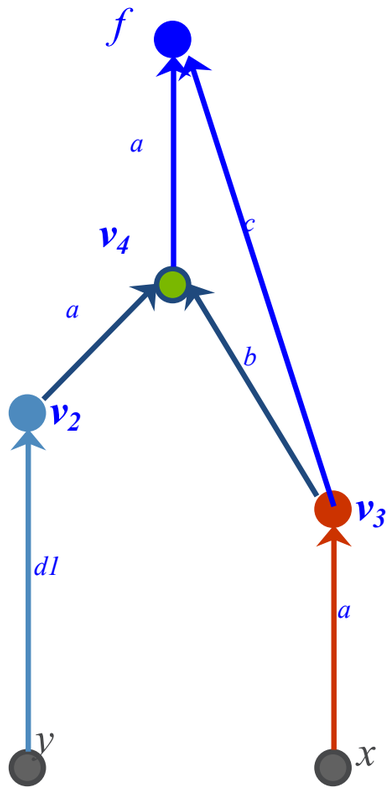
Forward mode: eliminate vertices in topological order



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

A GRAPH MODEL OF AD

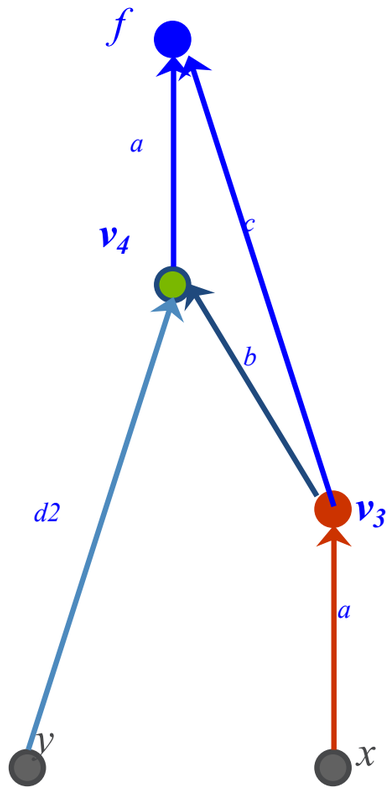
Forward mode: eliminate vertices in topological order



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

A GRAPH MODEL OF AD

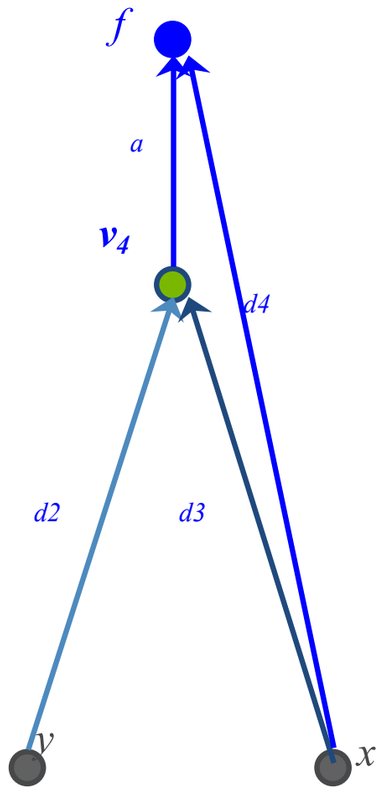
Forward mode: eliminate vertices in topological order



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

A GRAPH MODEL OF AD

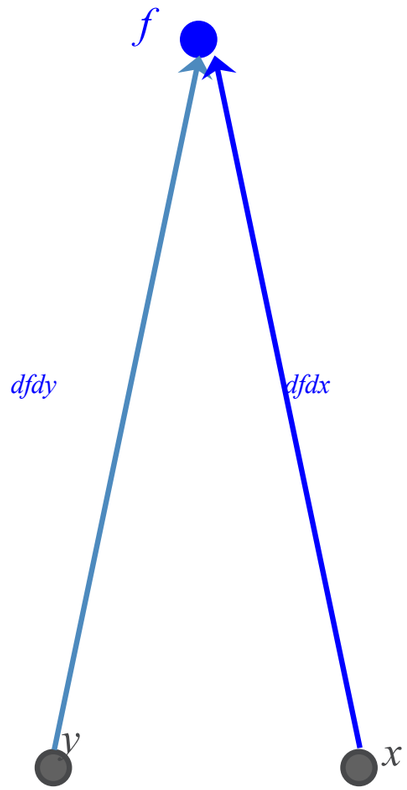
Forward mode: eliminate vertices in topological order



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

A GRAPH MODEL OF AD

Forward mode: eliminate vertices in topological order

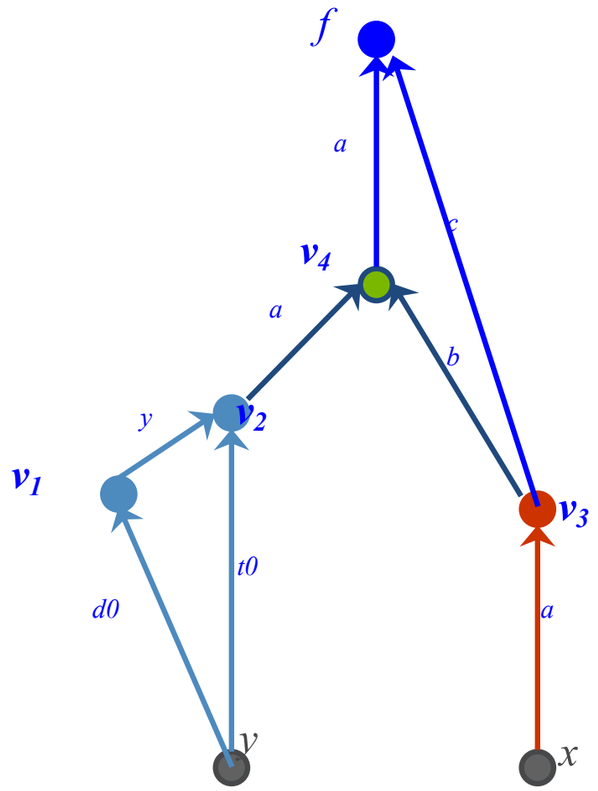



```
t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y
d2 = d1*a
d3 = a*b
d4 = a*c
dfdy = d2*a
dfdx = d4 + d3*a
```

6 mults 2 adds

A GRAPH MODEL OF AD

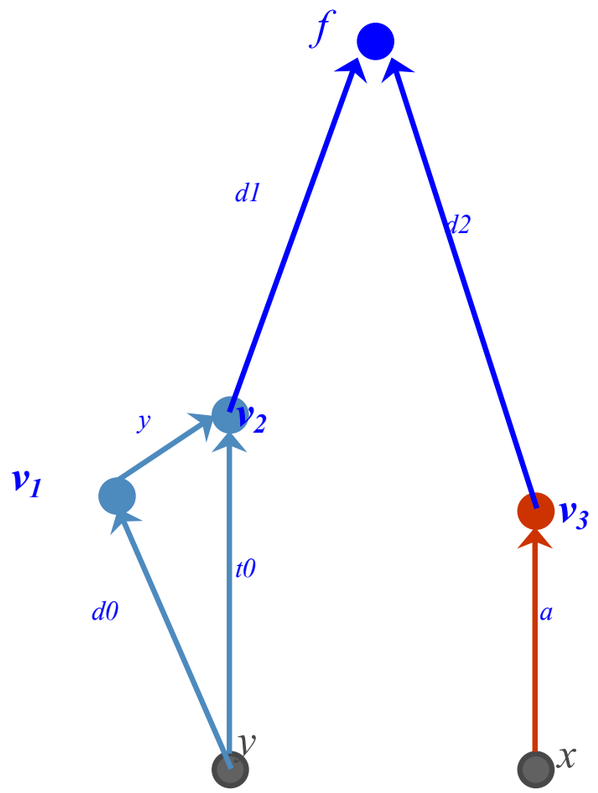
Reverse mode: eliminate vertices in reverse topological order



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

A GRAPH MODEL OF AD

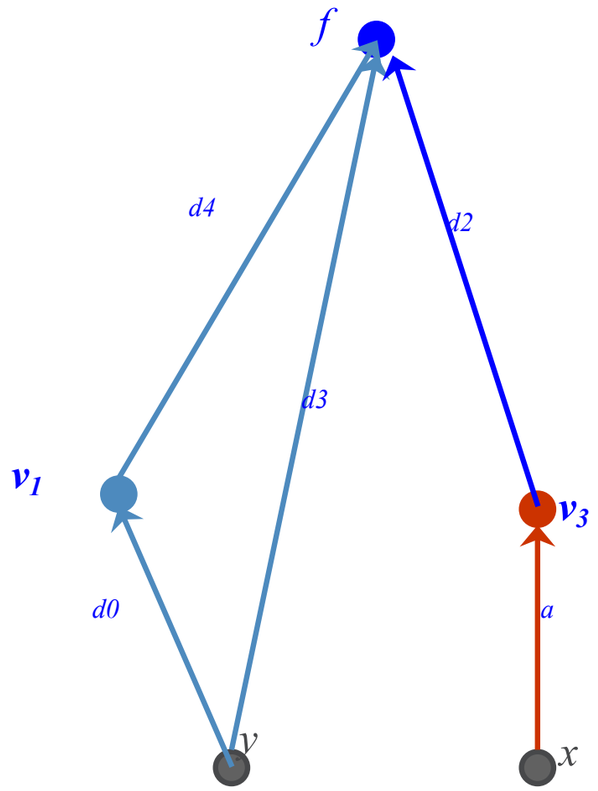
Reverse mode: eliminate vertices in reverse topological order



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

A GRAPH MODEL OF AD

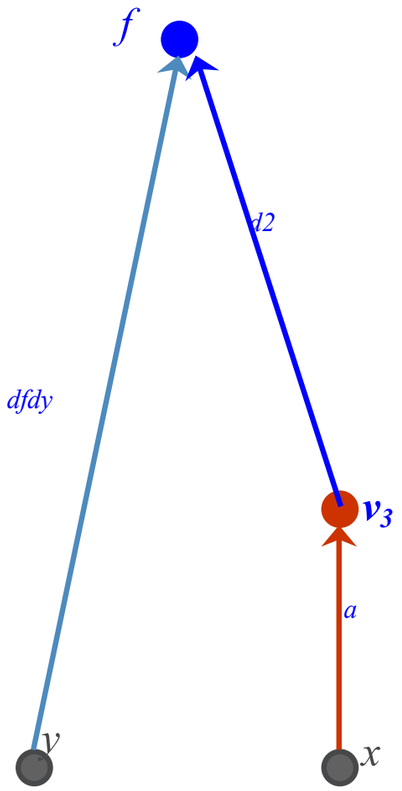
Reverse mode: eliminate vertices in reverse topological order



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

A GRAPH MODEL OF AD

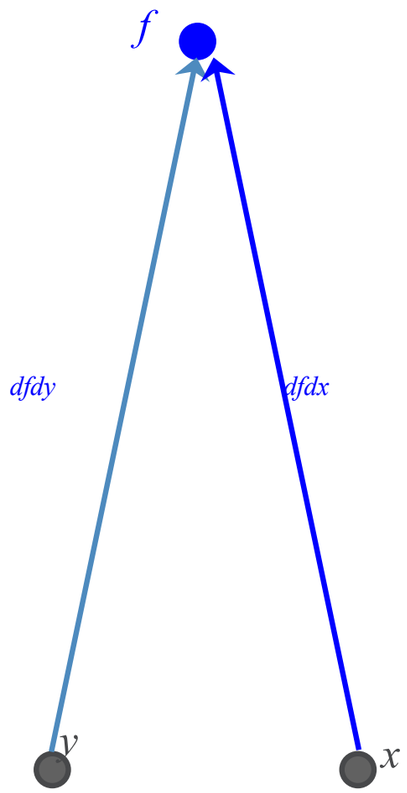
Reverse mode: eliminate vertices in reverse topological order



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

A GRAPH MODEL OF AD

Reverse mode: eliminate vertices in reverse topological order

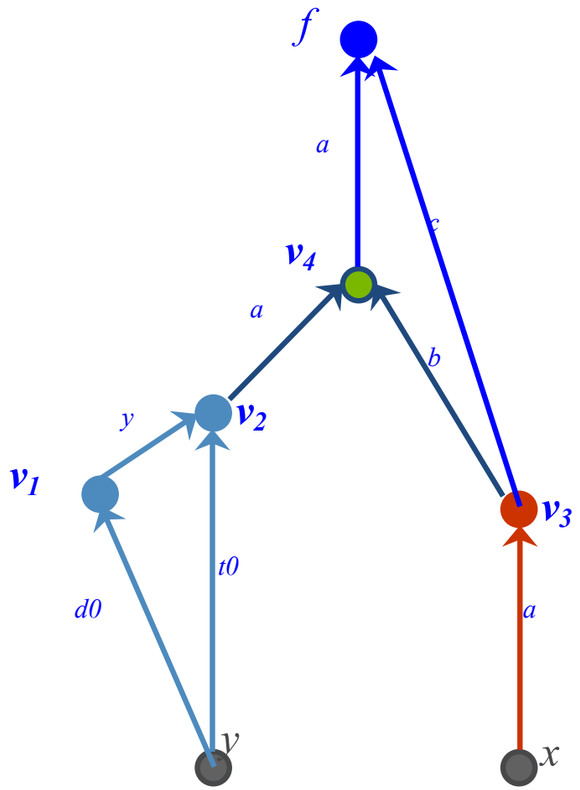


$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$
$dfdy = d3 + d0*d4$
$dfdx = a*d2$

6 mults 2 adds

A GRAPH MODEL OF AD

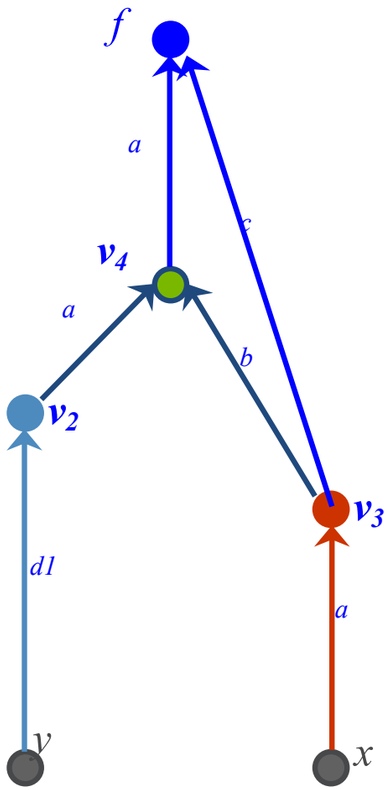
“Cross country” mode



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

A GRAPH MODEL OF AD

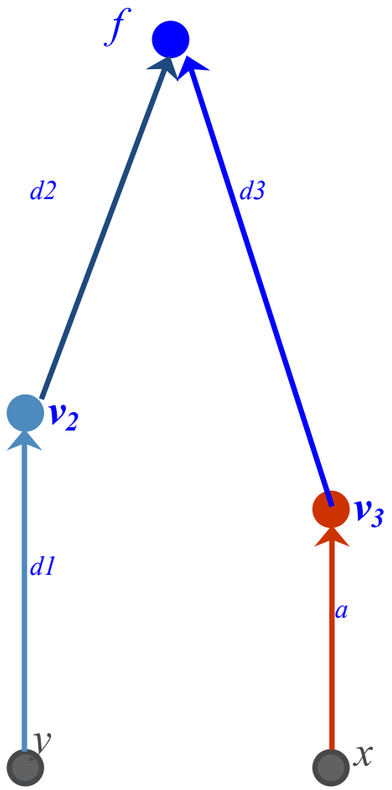
“Cross country” mode



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

A GRAPH MODEL OF AD

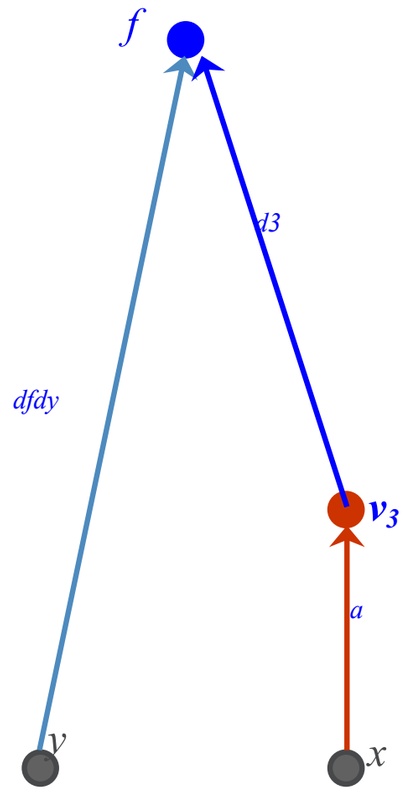
“Cross country” mode



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

A GRAPH MODEL OF AD

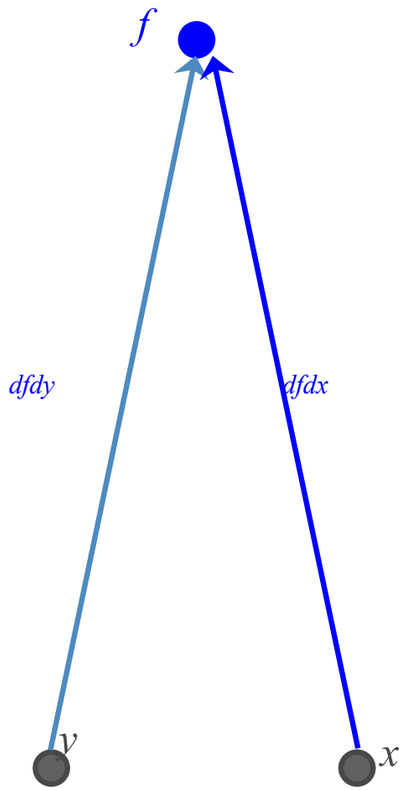
“Cross country” mode



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

A GRAPH MODEL OF AD

“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

PREACCUMULATION AND SCARCITY

Statement-Level Preaccumulation in ADIFOR

- My first project at Argonne (1991)
- Use forward mode as overall strategy, but differentiate each statement using the reverse mode
- Arose from recognition by Bischof and Griewank that implementing pure forward mode would require allocation of temporary arrays
- Reduces memory requirements and, frequently, number of operations

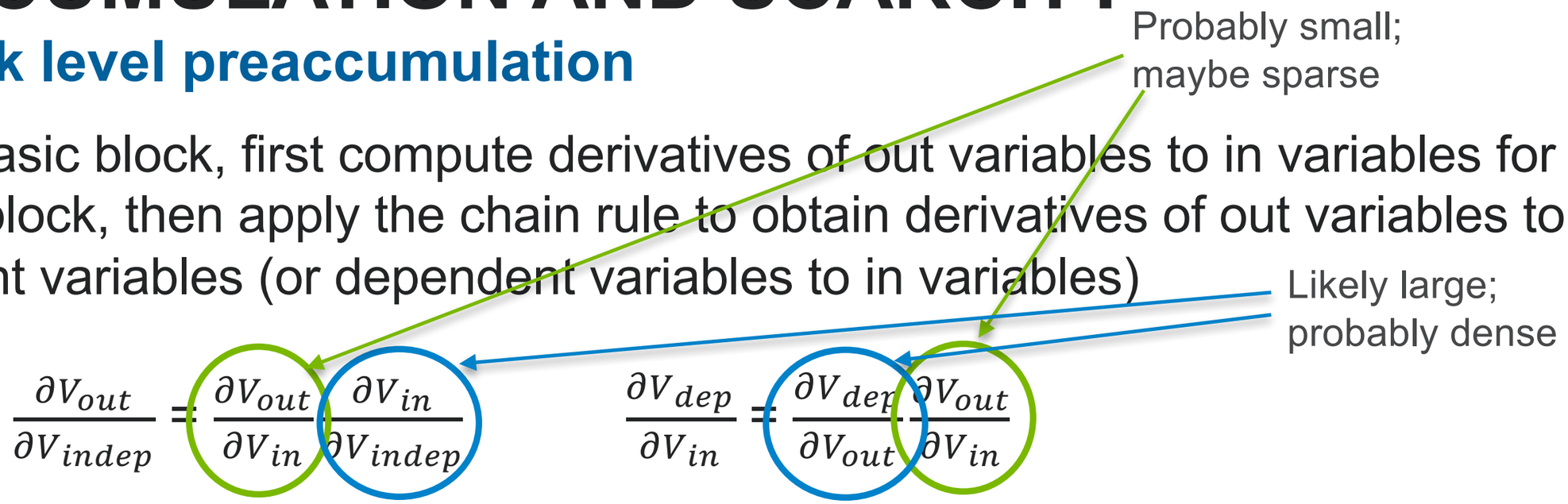
$$f(x) = x(1) * x(2) * x(3) * x(4) * x(5)$$

```
r$1 = x(1) * x(2)
r$2 = r$1 * x(3)
r$3 = r$2 * x(4)
r$4 = x(5) * x(4)
r$5 = r$4 * x(3)
r$1bar = r$5 * x(2)
r$2bar = r$5 * x(1)
r$3bar = r$4 * r$1
r$4bar = x(5) * r$2

do g$i$ = 1, g$p$
  g$f(g$i$) = r$1bar * g$x(g$i$, 1) + r$2bar * g$x(g$i$, 2)
              + r$3bar * g$x(g$i$, 3) + r$4bar * g$x(g$i$, 4)
              + r$3 * g$x(g$i$, 5)
end do
f = r$3 * x(5)
```

PREACCUMULATION AND SCARCITY

Basic-block level preaccumulation

- For each basic block, first compute derivatives of out variables to in variables for that basic block, then apply the chain rule to obtain derivatives of out variables to independent variables (or dependent variables to in variables)

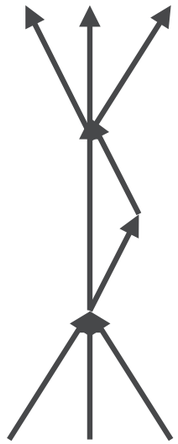


- In context of overall reverse mode strategy, offers potential to reduce the memory requirements for each basic block (store partial derivatives instead of intermediate variables)
- Storage and reverse mode accumulation cost is proportional to number of nonzeros in preaccumulated Jacobian

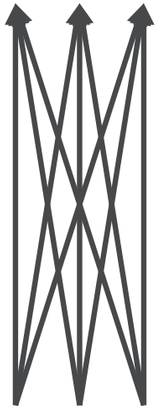
PREACCUMULATION AND SCARCITY

Minimal representation of a Jacobian (scarcity)

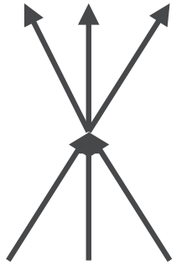
Original DAG



Bipartite DAG



Minimal DAG

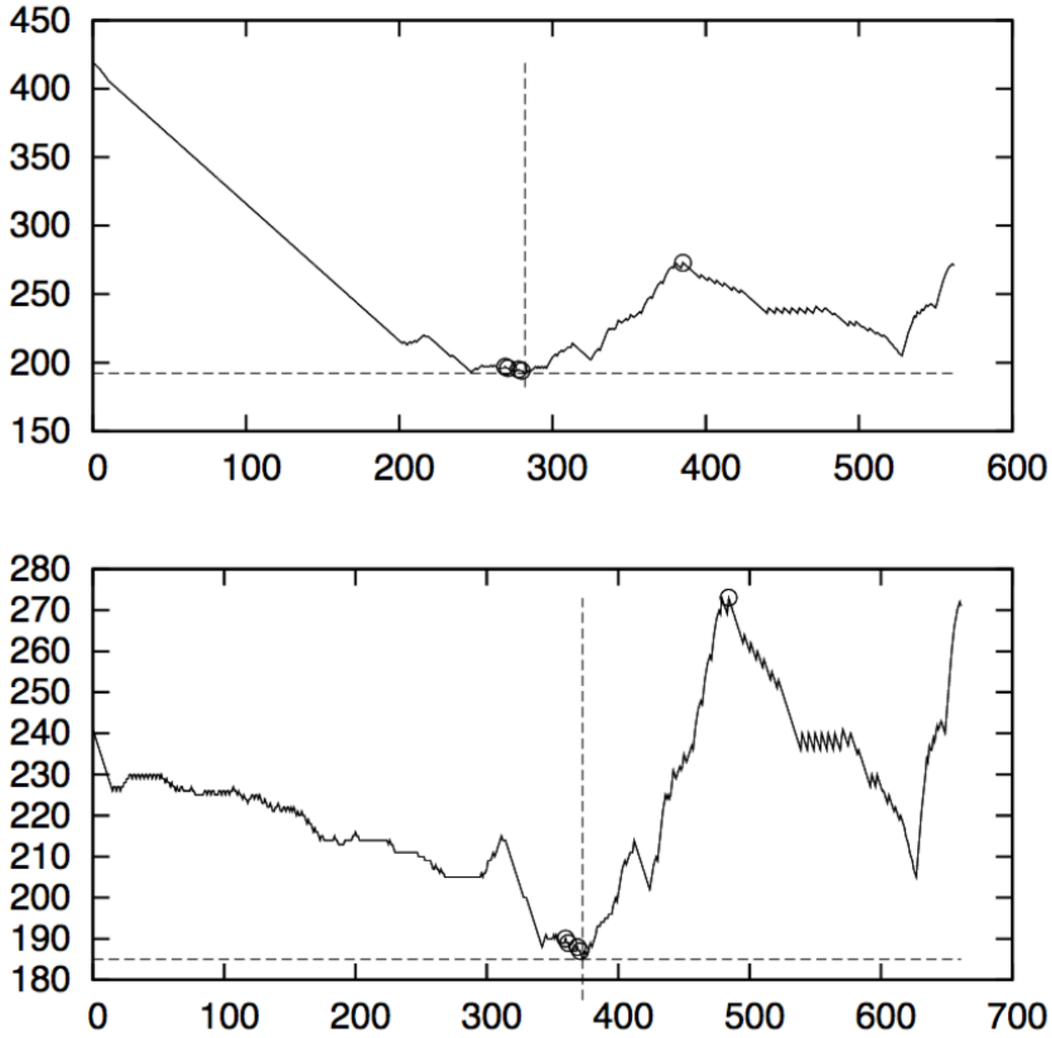


Reduce graph to one with minimal number of edges (or smallest number of DOF)
Avoid "catastrophic fill in" (empirical evidence that this happens in practice)
In essence, represent Jacobian as sum/product of sparse/low-rank matrices
Storage and accumulation costs proportional to number of edges

PREACCUMULATION AND SCARCITY

Prior work

- Heuristics for finding minimal representation
  - Apply some vertex elimination heuristic
  - Keep track of minimal intermediate graph
- IP formulation for minimal flop problem
  - Primary motivation: evaluate effectiveness of heuristics
  - Secondary motivation: find optimal order for key computational kernels
  - Jieqiu Chen et al.: AD2012



PREACCUMULATION AND SCARCITY

IP Formulation of Minimal Representation

- $x_{ijk} = 1$: there is an edge from vertex i to vertex j after step k
- $v_{ik} = 1$: vertex i is eliminated at step k
- Eliminate a vertex no more than once and eliminate no more than one vertex per step
- If an edge existed at the previous step, it must be preserved, unless the source or sink vertex is eliminated
- If a vertex is eliminated, then edge must exist from its predecessors to its successors

$$\sum_i v_{ik} \leq 1, \sum_k v_{ik} \leq 1$$

$$x_{ijk} \geq x_{ijk-1} - v_{ik} - v_{jk}$$

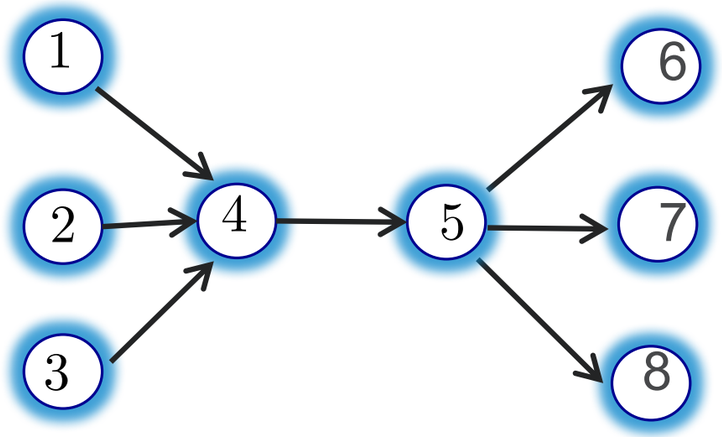
$$x_{ijk} \geq x_{ilk-1} + x_{ljk-1} + v_{lk} - 2$$

GAMS MODEL

```
binary variable x(i,j,k) "x[i,j,k] = 1 means edge (i,j) exists after turn k"
binary variable v(i,k) "v[i,k] = 1 means vertex i is eliminated at turn k"

Equations fa(i,j,k) graph at turn 1 must match givens
        fb(k) must eliminate no more than one vertex at each step
        fg(i) cannot eliminate the same vertex twice
        fc(k) do not eliminate independents
        fd(k) do not eliminate dependents
        fe(i,j,k) edge must be preserved unless source or sink is eliminated
        ff(i,j,k,l) edge must be introduced between predecessors and successors of eliminated vertices;

fa(i,j,k)$(given[i,j] and (ord(k) eq 1)..      x[i,j,k] =e= given(i,j);
fb(k)$(ord(k) gt 1)..                                  sum(i, v[i,k]) =l= 1;
fg(i)..                                                sum(k$(ord(k) gt 1), v[i,k]) =l= 1;
fc(k)..                                                sum(i$independent[i], v[i,k]) =e= 0;
fd(k)..                                                sum(i$dependent[i], v[i,k]) =e= 0;
fe(i,j,k)$(ord(k) gt 1)..                              x[i,j,k] =g= x[i,j,k-1] - v[i,k] - v[j,k];
ff(i,j,k,l)$(ord(k) gt 1)..                    x[i,j,k] =g= x[i,l,k-1] + x[l,j,k-1] + v[l,k] - 2;

variable obj; equation objdef; objdef.. obj =e= sum((i,j,k)$(ord(k) gt ninter), x[i,j,k]);

model minrep/ all /;
```

EXPERIMENTAL RESULTS

Example 1



Bipartite: 9

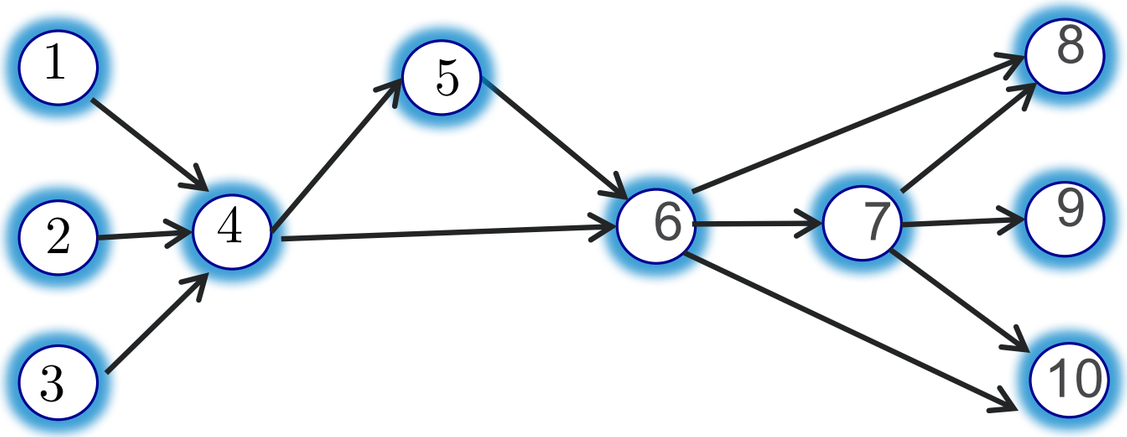
Forward minimum: 6

Reverse minimum: 6

IP minimum: 6

EXPERIMENTAL RESULTS

Example 2



Bipartite: 9

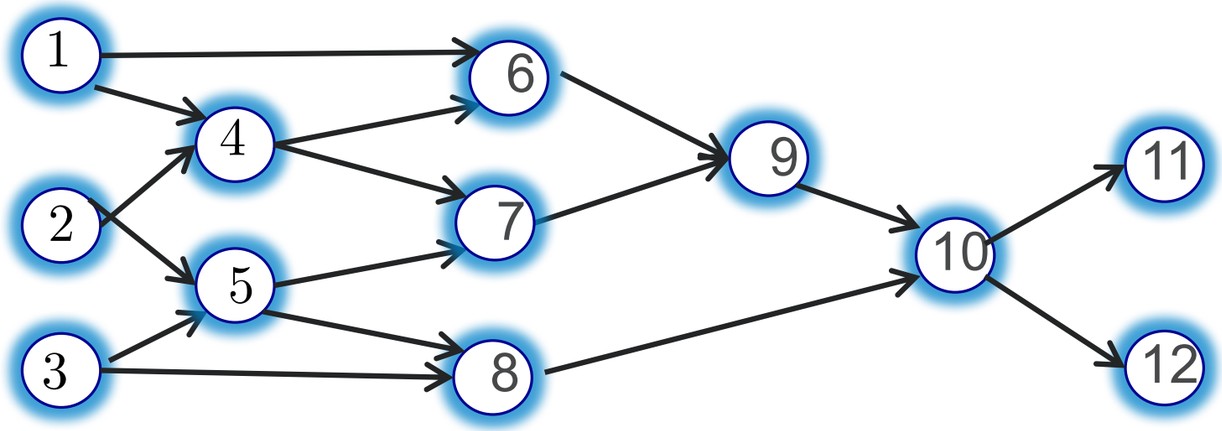
Forward minimum: 9

Reverse minimum: 6

IP minimum: 6

EXPERIMENTAL RESULTS

Example 3



Bipartite: 6

Forward minimum: 5

Reverse minimum: 6

IP minimum: 5

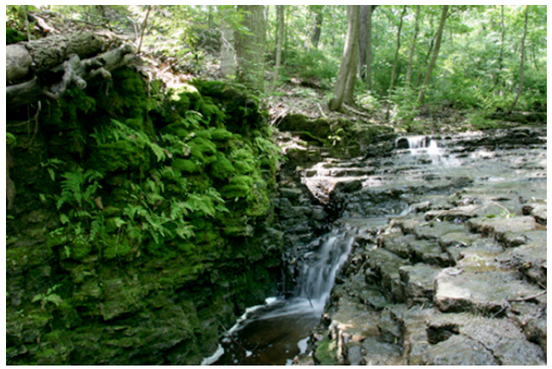
EXPERIMENTAL RESULTS

Graph	$\|v\|$	$\|v_{int}\|$	$\|e_s\|$	$\|e_f\|$	$\|e_b\|$	time
Example1	8	2	7	6	9	0.08s
Example2	10	4	12	6	9	0.09s
Example3	12	7	16	5	6	7.21s

CONCLUSIONS AND NEXT STEPS

- Storage and accumulation costs can be reduced by representing local Jacobian using the graph with the smallest number of edges (called scarcity by Griewank)
- Minimal representation problem can be modeled using integer programming
- IP solver finds optimal solution in seconds for small graphs
- Next steps
  - Extend model to use edge elimination
  - Integrate IP solve with XAIFBooster (more testcases)
  - Add bounds
  - Reduce size of state space (elimination step)
  - Find all minimal graphs

HELP WANTED!



- We anticipate having one or more postdoc openings soon
- We might have a staff opening soon
- We are always looking for short term and long term visitors (students and faculty)
- We are always looking for new collaborators

THANK YOU!

QUESTIONS?

THIS MATERIAL IS BASED UPON WORK SUPPORTED BY THE U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE, OFFICE OF ADVANCED SCIENTIFIC COMPUTING RESEARCH, APPLIED MATHEMATICS AND COMPUTER SCIENCE PROGRAMS UNDER CONTRACT DE-AC02-06CH11357