Higher-order derivatives, Jacobians, Beyond forward-reverse modes, and Randomized AD

UCSD CSE 291 Differentiable Programming

Tzu-Mao Li

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat
```

```
class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

what does this thing mean???

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat
```

```
class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

x.val.val: x's value
x.val.dval: the differential of the first differentiation
x.dval.val: the differential of the second differentiation
x.dval.dval: no particular meaning, should always be 0!!

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

let's try to generate Df & D2f by hand!

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

```
def Df(x : _dfloat, y : _dfloat) -> _dfloat:
  return make__dfloat(
      x.val * y.val,
      x.val * y.dval + x.dval * y.val
    )
```

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

```
def Df(x : _dfloat, y : _dfloat) -> _dfloat:
  return make__dfloat(
      x.val * y.val,
      x.val * y.dval + x.dval * y.val
    )
```

```
def D2f(x : _d_dfloat, y : _d_dfloat) -> \
    _d_dfloat:
  ret : _d_dfloat
  ret.val = make__dfloat(
      x.val.val * y.val.val,
      x.val.val * y.dval.val + x.dval.val * y.val.val
    )
  ret.dval = make__dfloat(
      x.val.val * y.val.dval + x.val.dval * y.val.val,
      x.val.val * y.dval.dval + x.val.dval * y.dval.val +
      x.dval.val * y.val.dval + x.dval.dval * y.val.val
    )
  return ret
```

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)
# Df(x : _dfloat, y : _dfloat) -> _dfloat

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

```
def Df(x : _dfloat, y : _dfloat) -> _dfloat:
  return make__dfloat(
      x.val * y.val,
      x.val * y.dval + x.dval * y.val
    )
```

actual second derivatives

(notice that x.dval.dval is multiplied with y.val.val — which should not appear in any second derivatives!)

```
def D2f(x : _d_dfloat, y : _d_dfloat) -> \
    _d_dfloat:
  ret : _d_dfloat
  ret.val = make__dfloat(
      x.val.val * y.val.val,
      x.val.val * y.dval.val + x.dval.val * y.val.val
    )
  ret.dval = make__dfloat(
      x.val.val * y.val.dval + x.val.dval * y.val.val,
      x.val.val * y.dval.dval + x.val.dval * y.dval.val +
      x.dval.val * y.val.dval + x.dval.dval * y.val.val
    )
  return ret
```

Nested forward mode application

```
def f(x : float, y : float) -> float:
  return x * y

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

$$\frac{\partial^2 f}{\partial x^2} = 0$$

set x.val.dval = 1, x.dval.val = 1
others to 0

$$\frac{\partial^2 f}{\partial y^2} = 0$$

set y.val.dval = 1, y.dval.val = 1
others to 0

```
x.val.val * y.dval.dval + x.val.dval * y.dval.val +
x.dval.val * y.val.dval + x.dval.dval * y.val.val
```

x.val.val: x's value
x.val.dval: diff. of the first diff.
x.dval.val: diff. of the second diff.
x.dval.dval: should always be 0!!

Nested forward mode application

quiz: when will $\frac{\partial}{\partial x}\frac{\partial f}{\partial y} \neq \frac{\partial}{\partial y}\frac{\partial f}{\partial x}$?

```
def f(x : float, y : float) -> float:
  return x * y

D2f = fwd_diff(Df)
# D2f(x : _d_dfloat, y : _d_dfloat) -> _d_dfloat

class _d_dfloat:
  val : _dfloat
  dval : _dfloat
```

```
x.val.val * y.dval.dval + x.val.dval * y.dval.val +
x.dval.val * y.val.dval + x.dval.dval * y.val.val
```

```
x.val.val: x's value
x.val.dval: diff. of the first diff.
x.dval.val: diff. of the second diff.
x.dval.dval: should always be 0!!
```

$$\frac{\partial^2 f}{\partial x^2} = 0$$

set x.val.dval = 1, x.dval.val = 1
others to 0

$$\frac{\partial^2 f}{\partial y^2} = 0$$

set y.val.dval = 1, y.dval.val = 1
others to 0

$$\frac{\partial}{\partial x}\frac{\partial f}{\partial y} = 1$$

set y.val.dval = 1, x.dval.val = 1
others to 0

$$\frac{\partial}{\partial y}\frac{\partial f}{\partial x} = 1$$

set x.val.dval = 1, y.dval.val = 1
others to 0

A better way to do nested differentiation

(to avoid passing the wrong differentials)

```
def f(x : float, y : float) -> float:
  return x * y

Df = fwd_diff(f)

def dfdx(x : float, y : float) -> float:
  return Df(make__dfloat(x, 1),
            make__dfloat(y, 0)).dval

Ddfdx = fwd_diff(dfdx)

def d2fdx2(x : float, y : float) -> float:
  return Ddfdx(make__dfloat(x, 1),
               make__dfloat(y, 0)).dval
```

One detail: perturbation confusion

a common bug in functional differentiable programming!

$$\frac{\partial}{\partial x}\left(x \cdot \left(\frac{\partial}{\partial y}x + y\Big|_{y=1}\right)\right)\Bigg|_{x=1}$$

when expanding this expression using dual number,
if we use the same ϵ for both x and y, the results would be incorrect
(demonstrate if we have time)
instead, we need to use different ϵ for the two variables

Perturbation Confusion and Referential Transparency

Correct Functional Implementation of Forward-Mode AD

Jeffrey Mark Siskind[1] and Barak A. Pearlmutter[2]

[1] School of Electrical and Computer Engineering, Purdue University, USA
[2] Hamilton Institute, NUI Maynooth, Ireland
[1] qobi@purdue.edu [2] barak@cs.nuim.ie

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

```
f: x -> y
```

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

forward mode

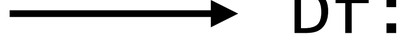
`f: x -> y` ⟶ `Df: x, dx -> dy`

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

forward mode

```
f: x -> y
```



```
Df: x, dx -> dy
```

reverse mode

```
DTDf: x, dx, d*dy -> d*x, d*dx
```

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

forward mode

`f: x -> y` ⟶ `Df: x, dx -> dy`

↓ reverse mode

`DTDf: x, dx, d*dy -> d*x, d*dx`

↓ set d*dy = 1
drop d*dx (doesn't particularly mean anything)

`Hvf: x, dx -> d*x`

$H \cdot dx$ "Hessian vector product"!

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

forward mode

`f: x -> y` ⟶ `Df: x, dx -> dy`

since both forward and reverse modes preserve time complexity,
Hessian vector product preserves the complexity of the primal function!

↓ reverse mode

`DTDf: x, dx, d*dy -> d*x, d*dx`

↓ set d*dy = 1
↓ drop d*dx (doesn't particularly mean anything)

`Hvf: x, dx -> d*x`

$H \cdot dx$ "Hessian vector product"!

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

reverse mode

can also apply forward after reverse mode

`f: x -> y` ⟶ `DTf: x, d*y -> d*x`

↓ forward mode

`DDTf: x, dx, d*y, dd*y -> dd*x`

↓ set $d^*y = 1$ and $dd^*y = 0$

`Hvf: x, dx -> dd*x`

Hessian computation

$x : \mathbb{R}^n, y : \mathbb{R}$

reverse mode

can also apply forward after reverse mode

```
f: x -> y  ⟶  DTf: x, d*y -> d*x
```

forward mode

```
DDTf: x, dx, d*y, dd*y -> dd*x
```

set $d^*y = 1$ and $dd^*y = 0$

```
Hvf: x, dx -> dd*x
```

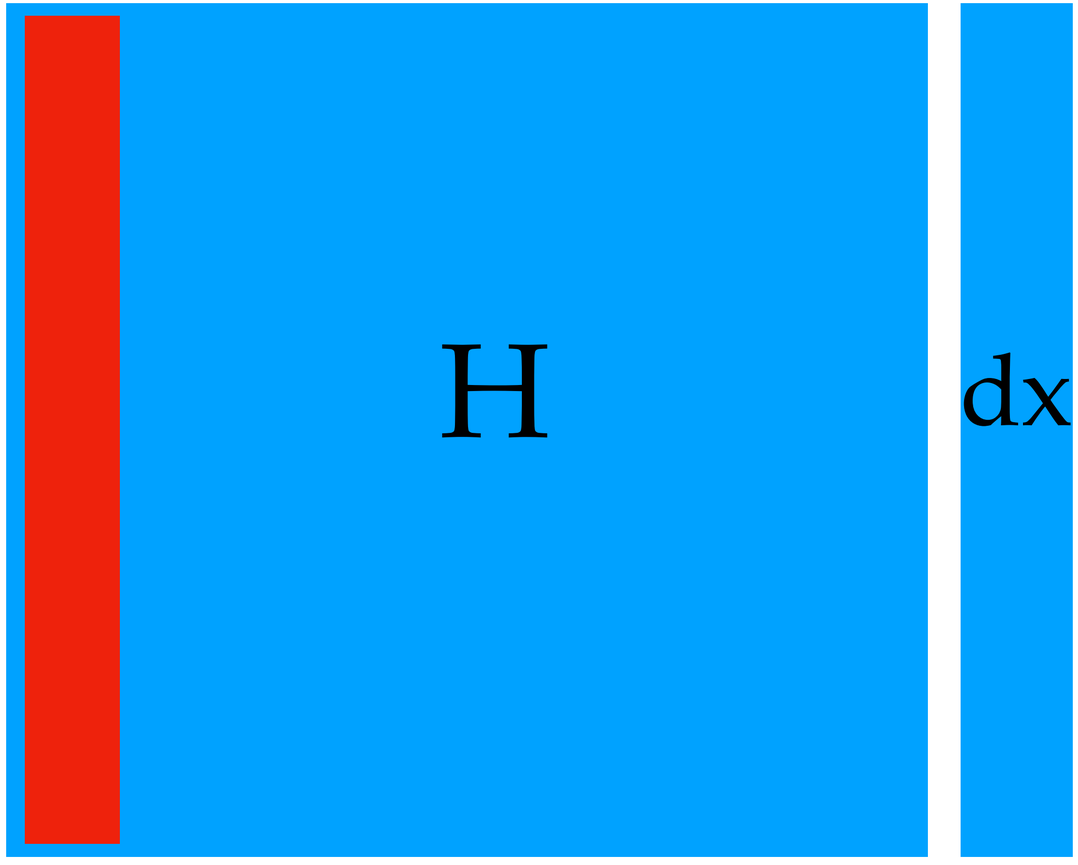
quiz: does applying reverse mode over reverse mode for Hessian computation make sense?

Assembling the Hessian

```
Hvf: x, dx -> d*x
```

let dx = [1, 0, 0, 0, 0, …]^T

each dx computes one column of H



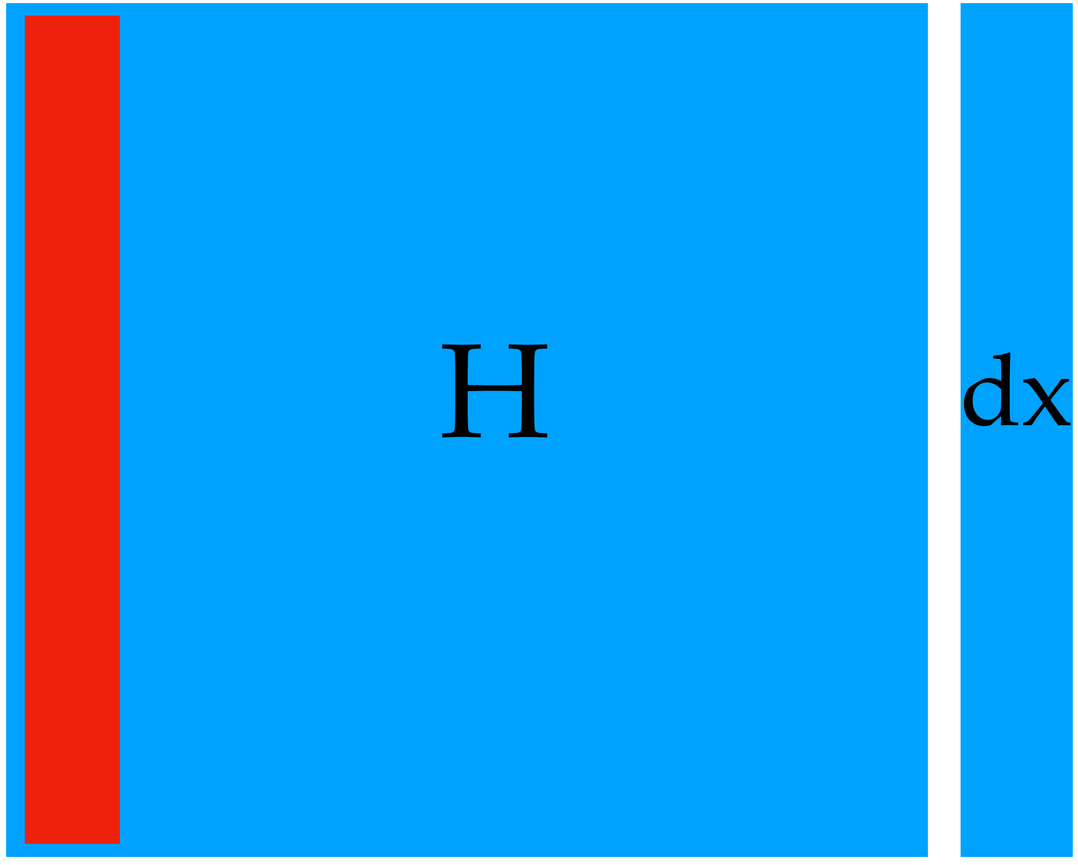
Assembling the Hessian

Hessian assembly takes O(n) time (n is the input dimensionality)

```
Hvf: x, dx -> d*x
```

let dx = [1, 0, 0, 0, 0, …]^T

each dx computes one column of H



Assembling the Hessian

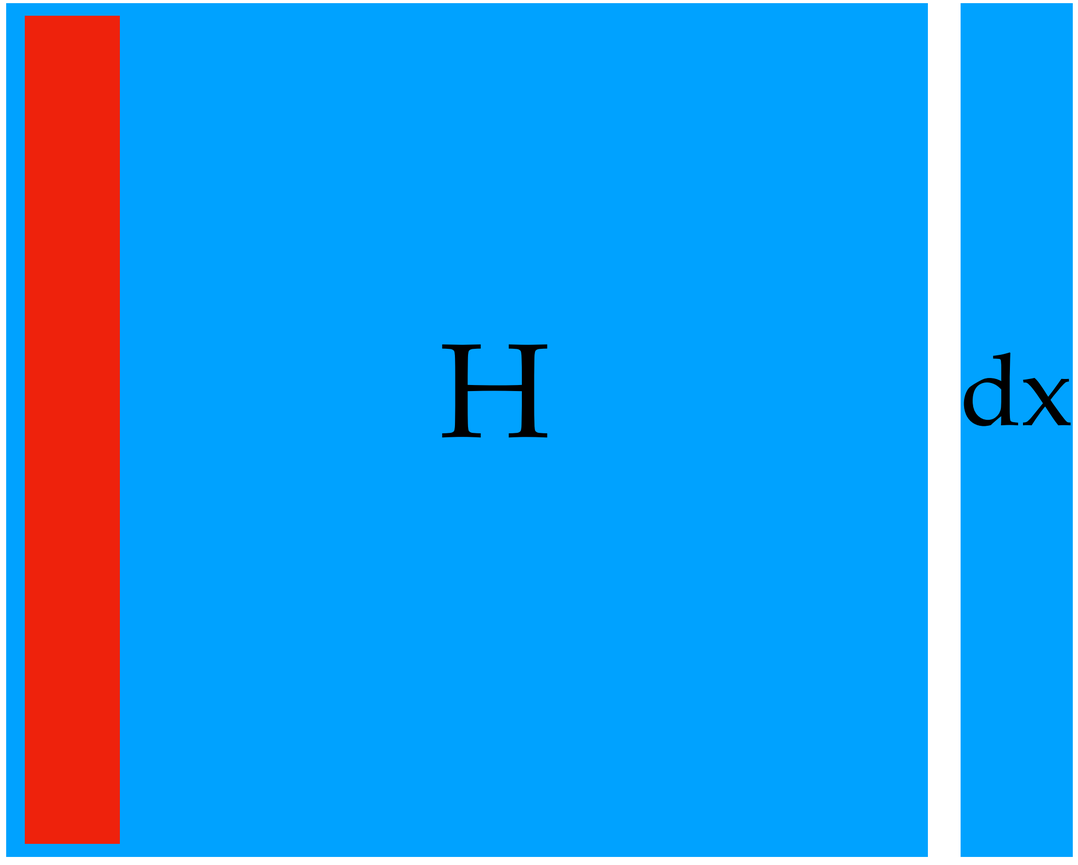
Hessian assembly takes O(n) time (n is the input dimensionality)

notice: this does at least ~twice amount of necessary computation, since H is almost always symmetric!

```
Hvf: x, dx -> d*x
```

let dx = [1, 0, 0, 0, 0, …]^T

each dx computes one column of H



Speeding up Hessian computation?

notice that Hessian is the Jacobian matrix of the gradient function

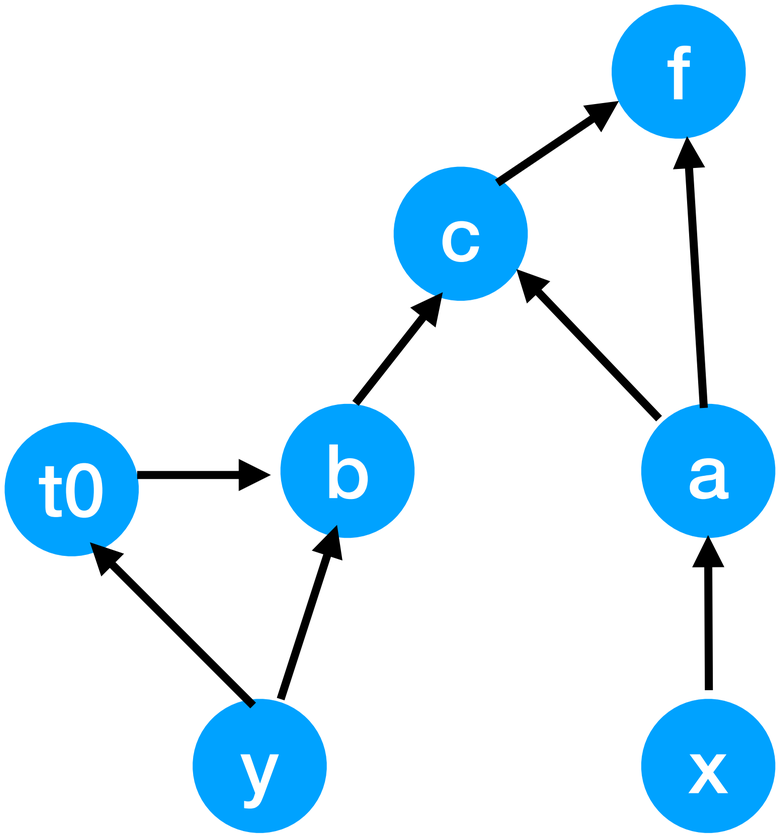
$$Hf = J\nabla f$$

if we have a way to quickly compute Jacobian, we have a way to speedup Hessian

to achieve this, we generalize forward and reverse mode into the **cross country mode**

Let's revisit the computational graph

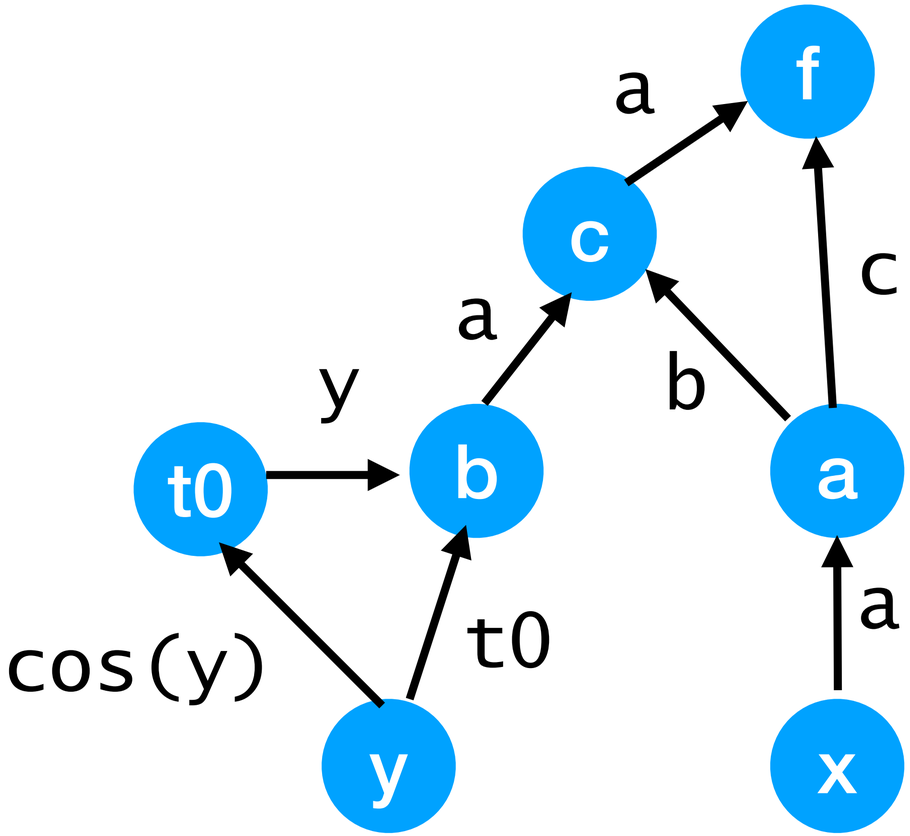
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Let's revisit the computational graph

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

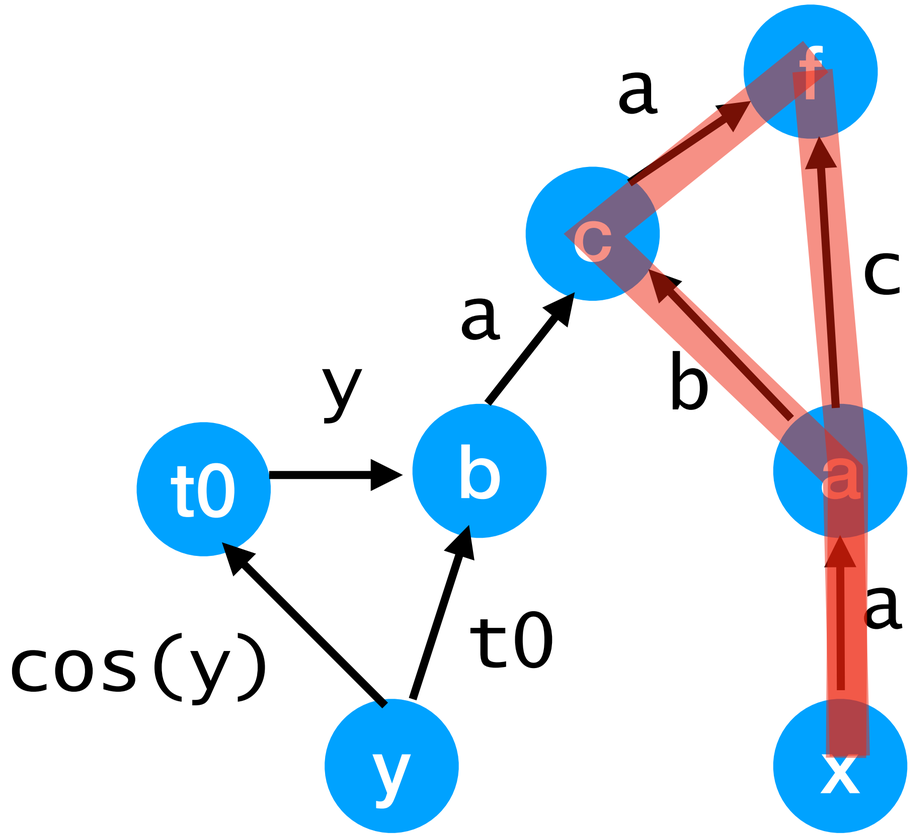


edge = partial derivatives

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Derivatives = counting paths from input to output

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



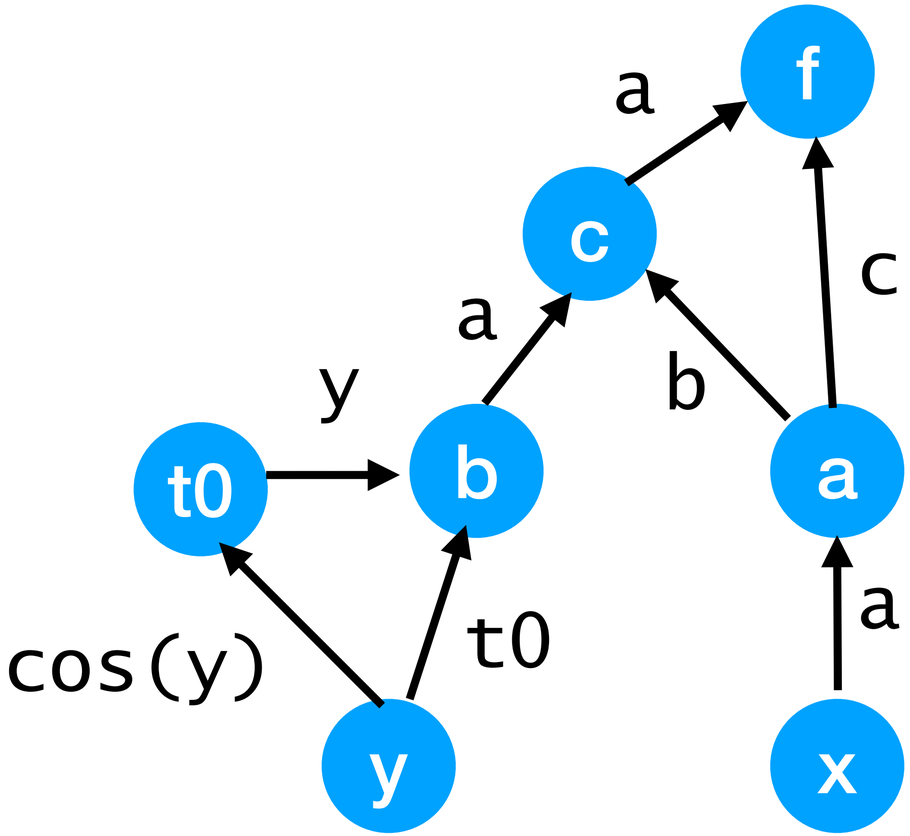
edge = partial derivatives

```
dfdx = a*c + a*b*a
```

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Vertex elimination: produce an equivalent graph by removing one vertex

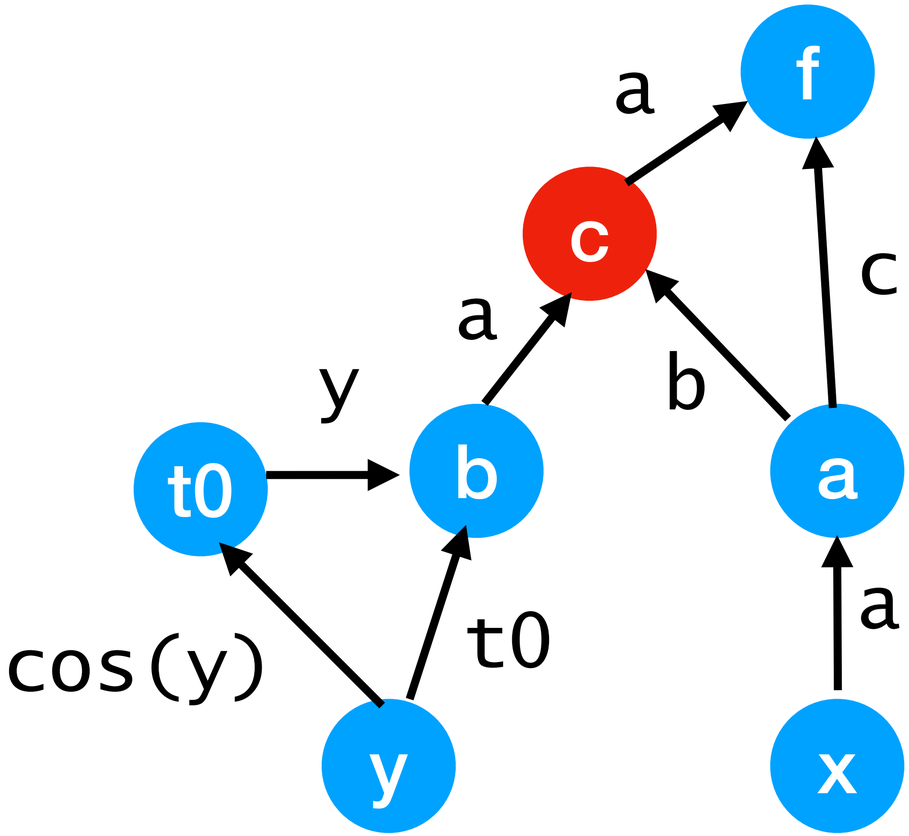
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Vertex elimination: produce an equivalent graph by removing one vertex

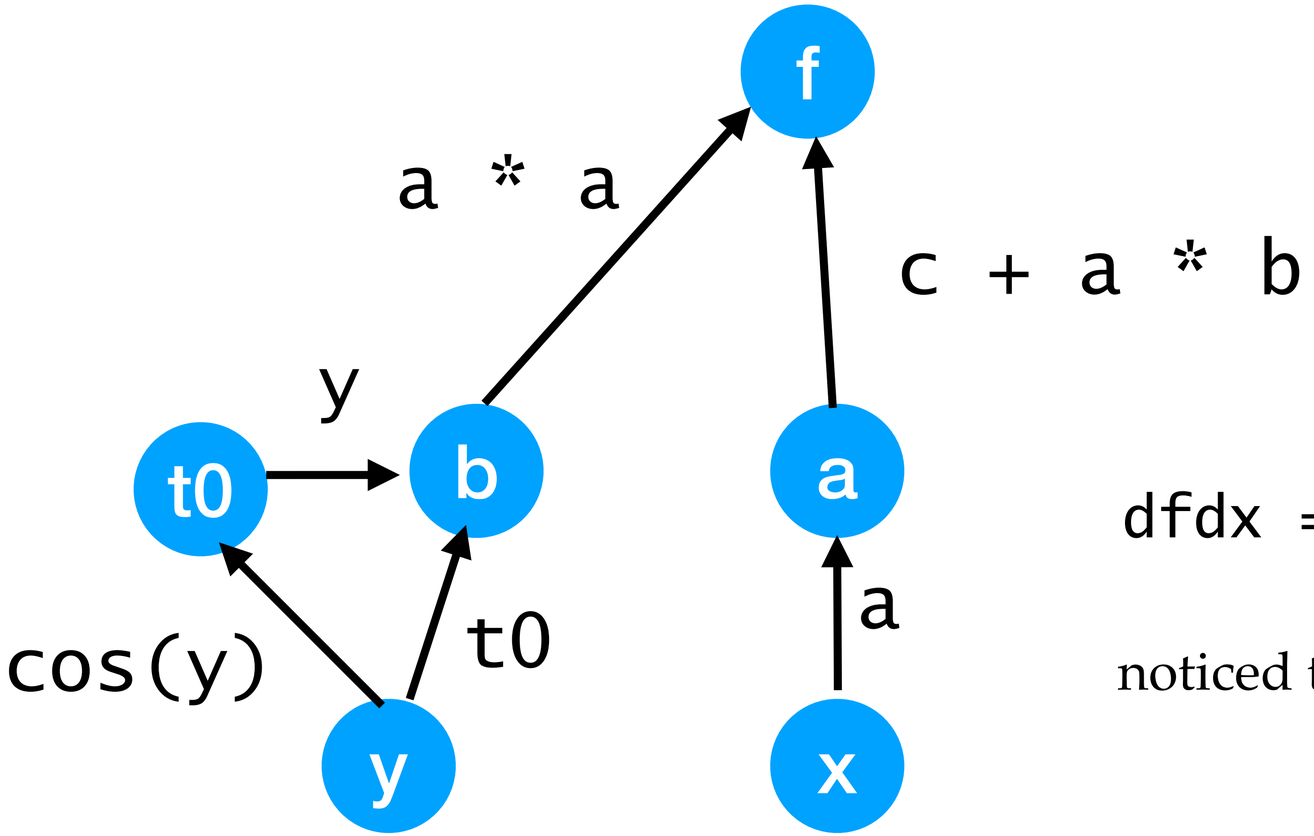
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Vertex elimination: produce an equivalent graph by removing one vertex

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



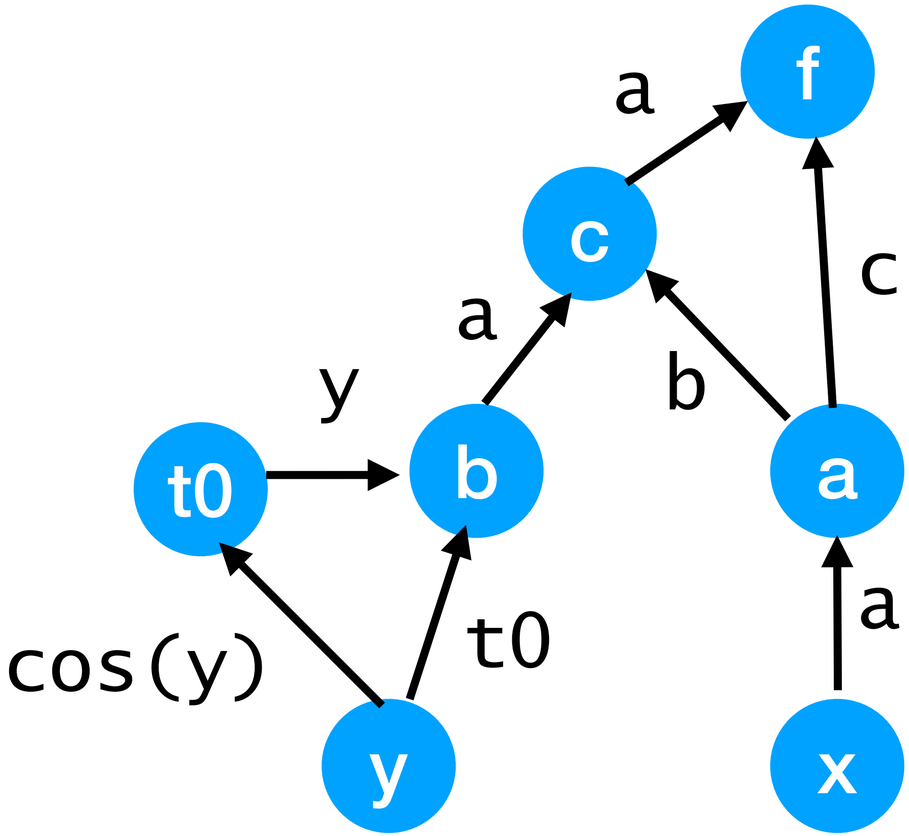

```
dfdx = a*(c + a*b)
```

noticed that a is factored out!!

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

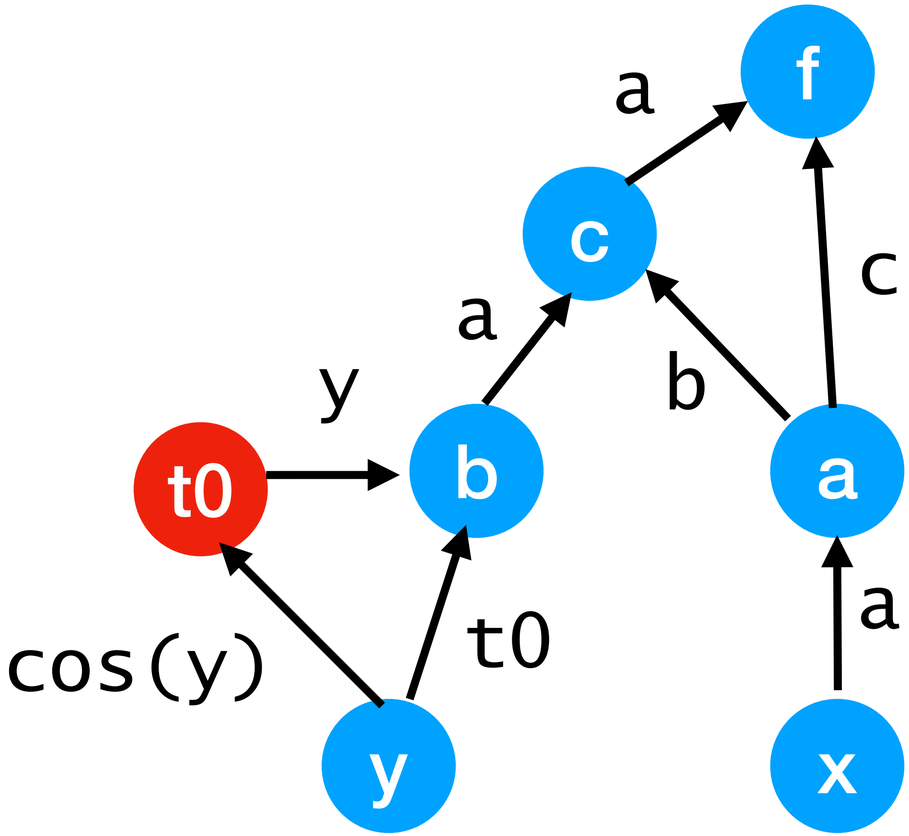
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

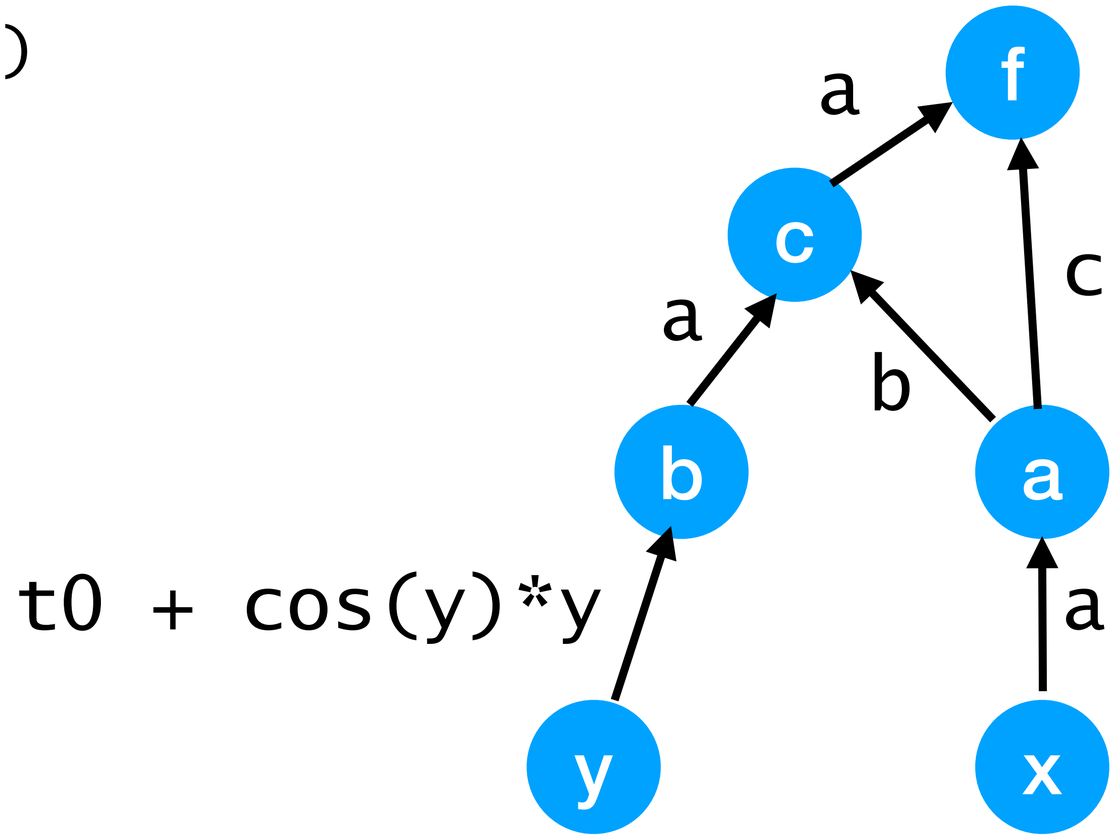
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

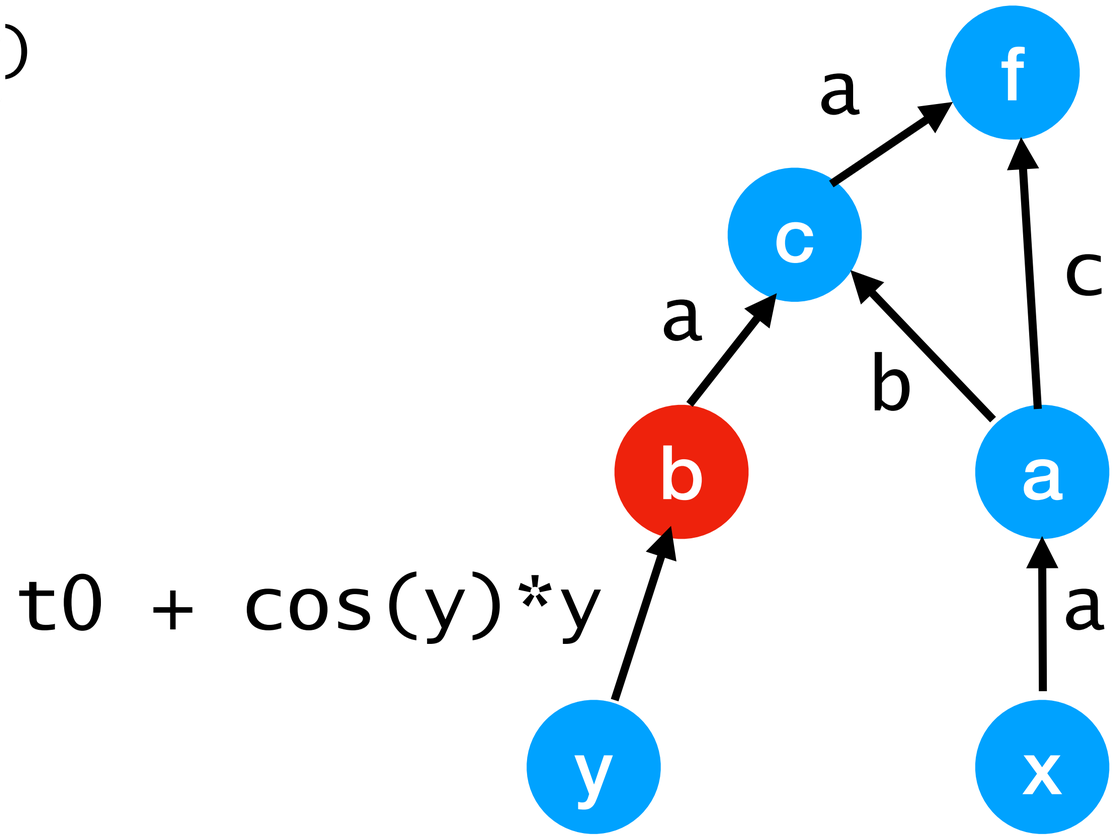
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

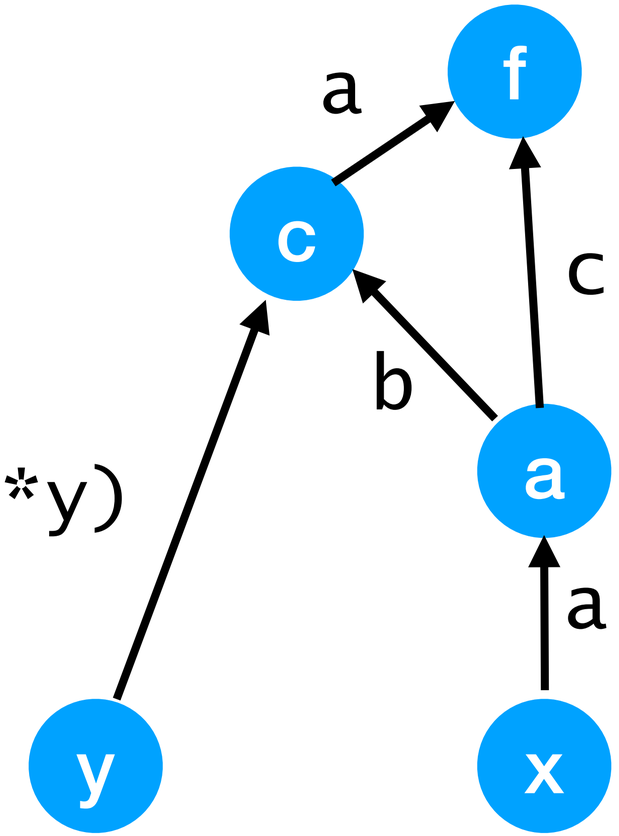


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)
```

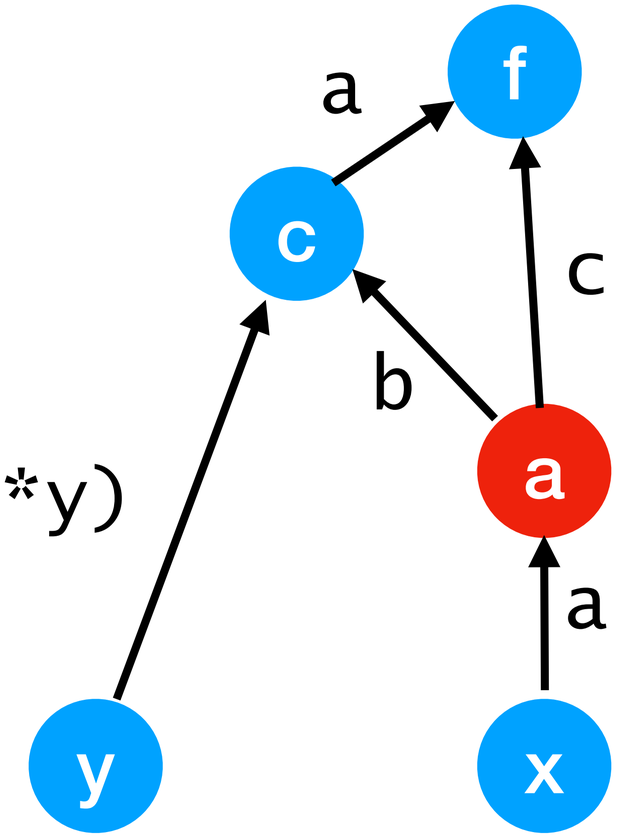


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)
```

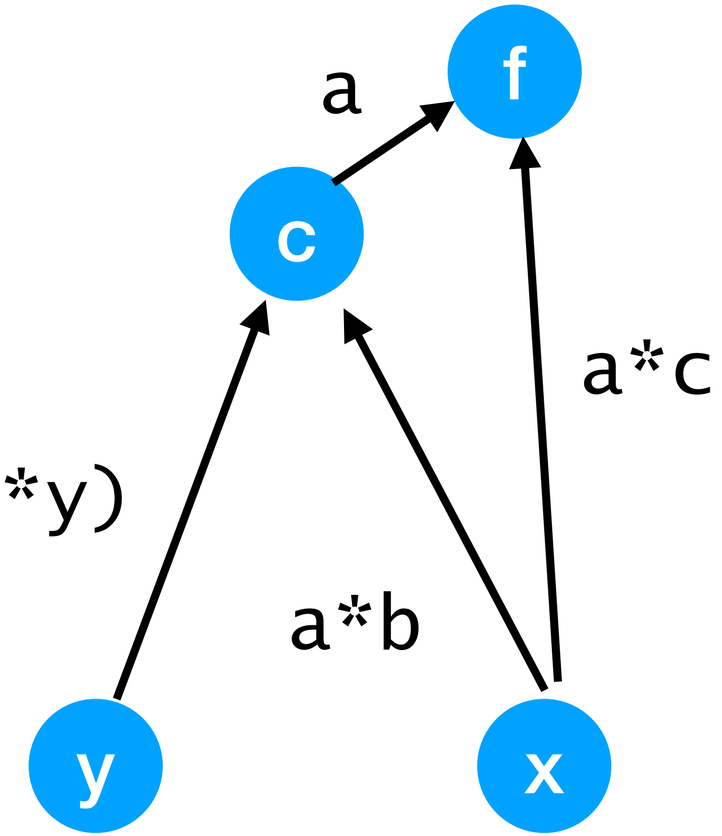


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)
```

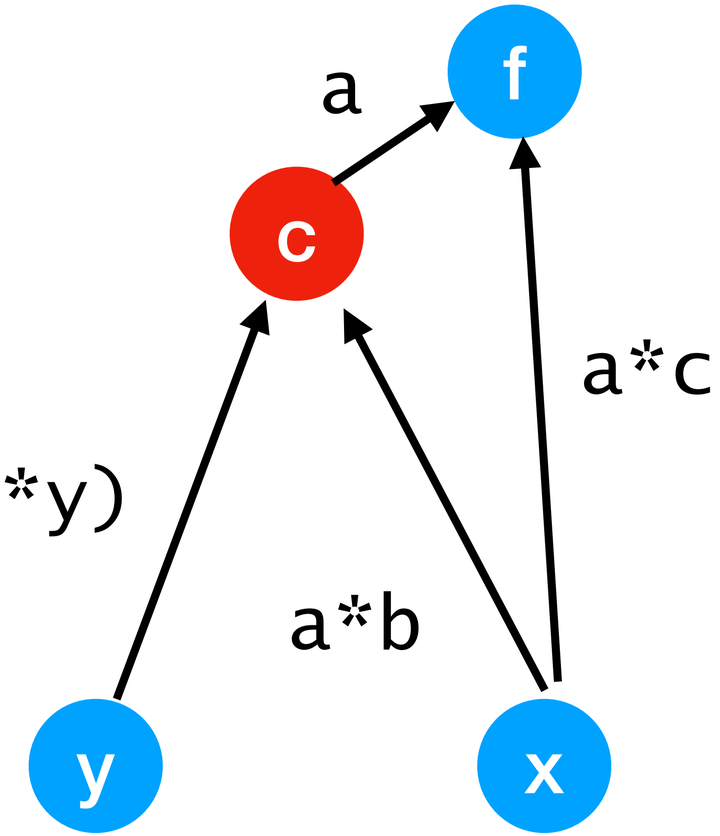


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)
```

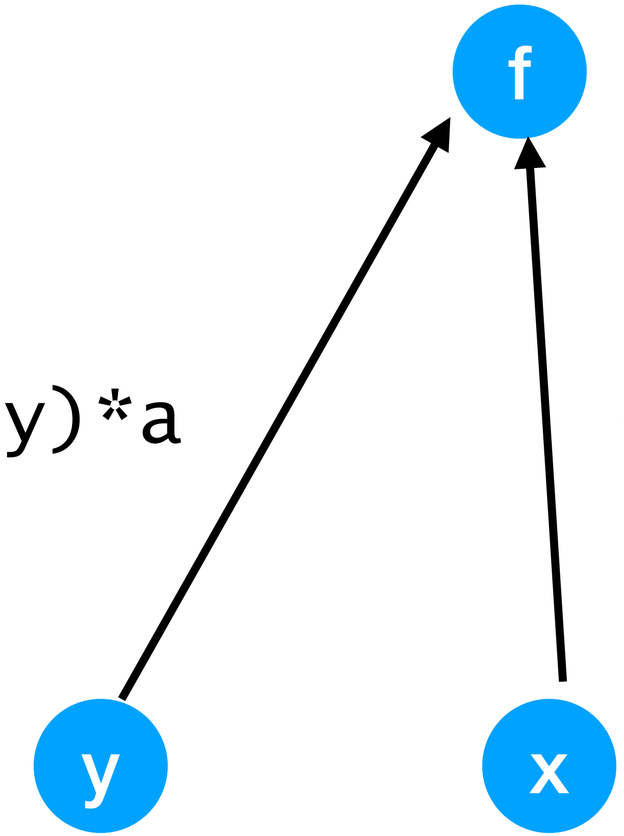


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)*a
```




```
a*c+a*a*b
```

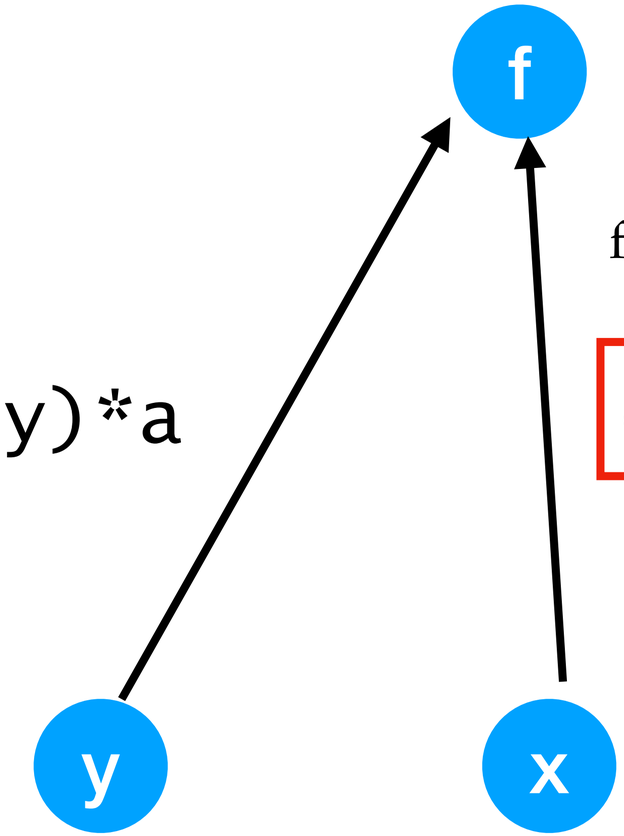
6 mults, 2 adds
(count the number of * and +)

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Forward mode: eliminate vertices from inputs to outputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

```
a*(t0 + cos(y)*y)*a
```



forward mode fails to factor out **a** here!

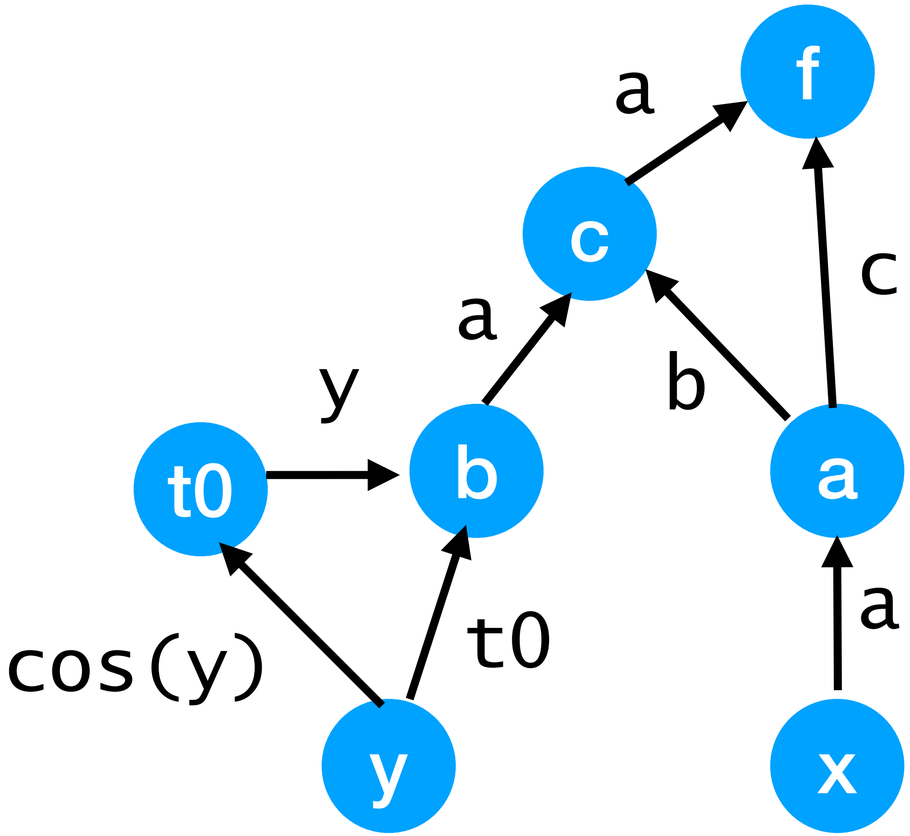
```
a*c+a*a*b
```

6 mults, 2 adds
(count the number of * and +)

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

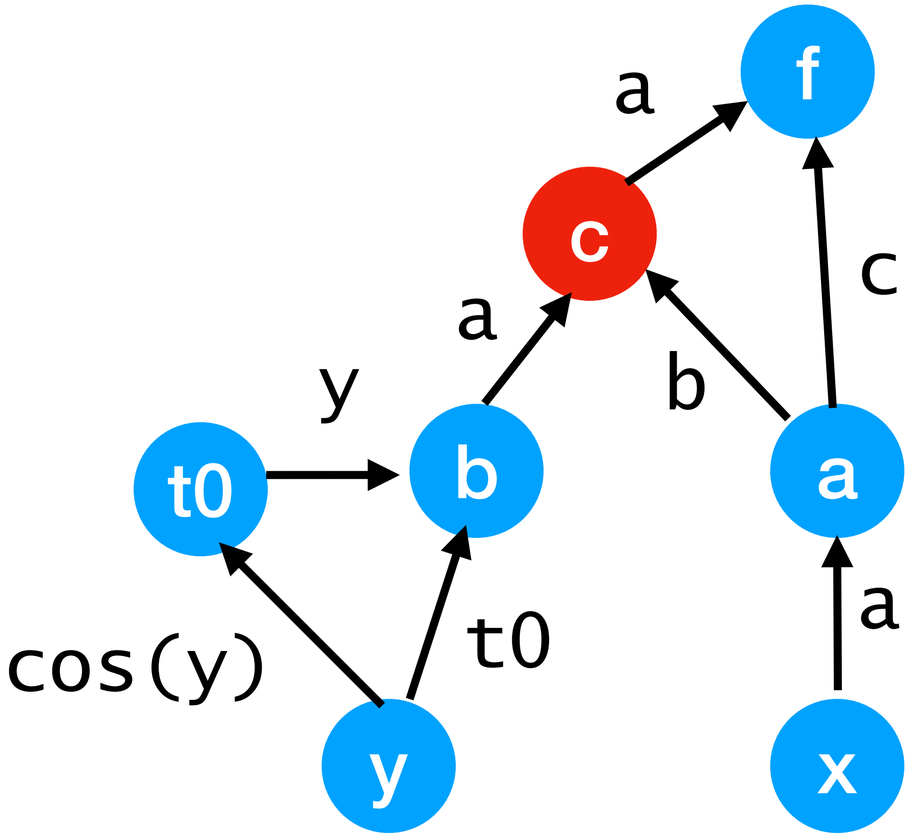
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

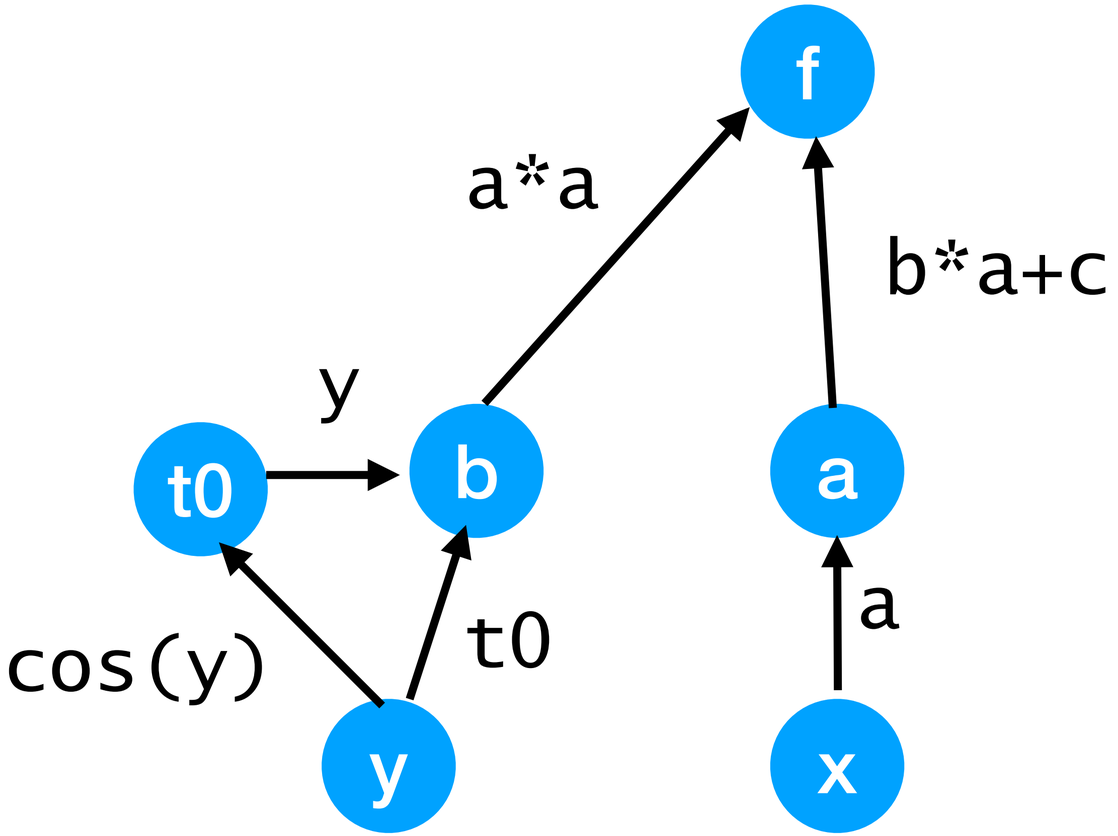
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

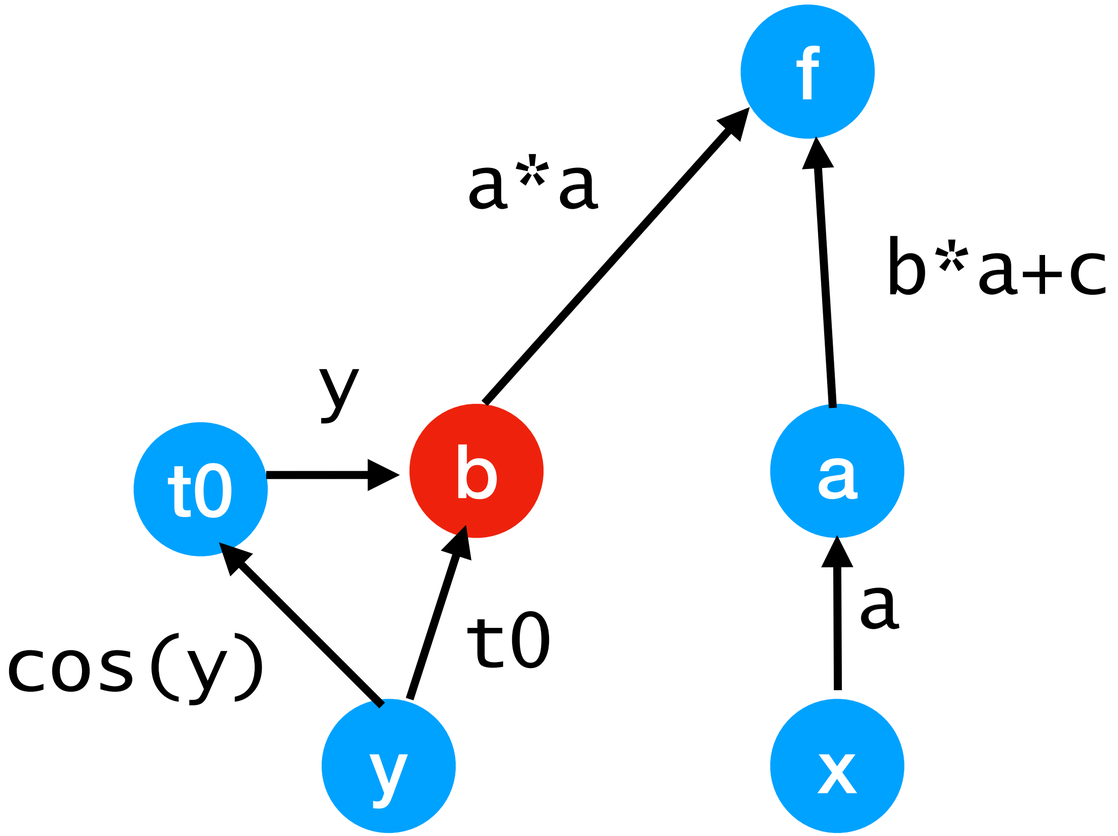
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

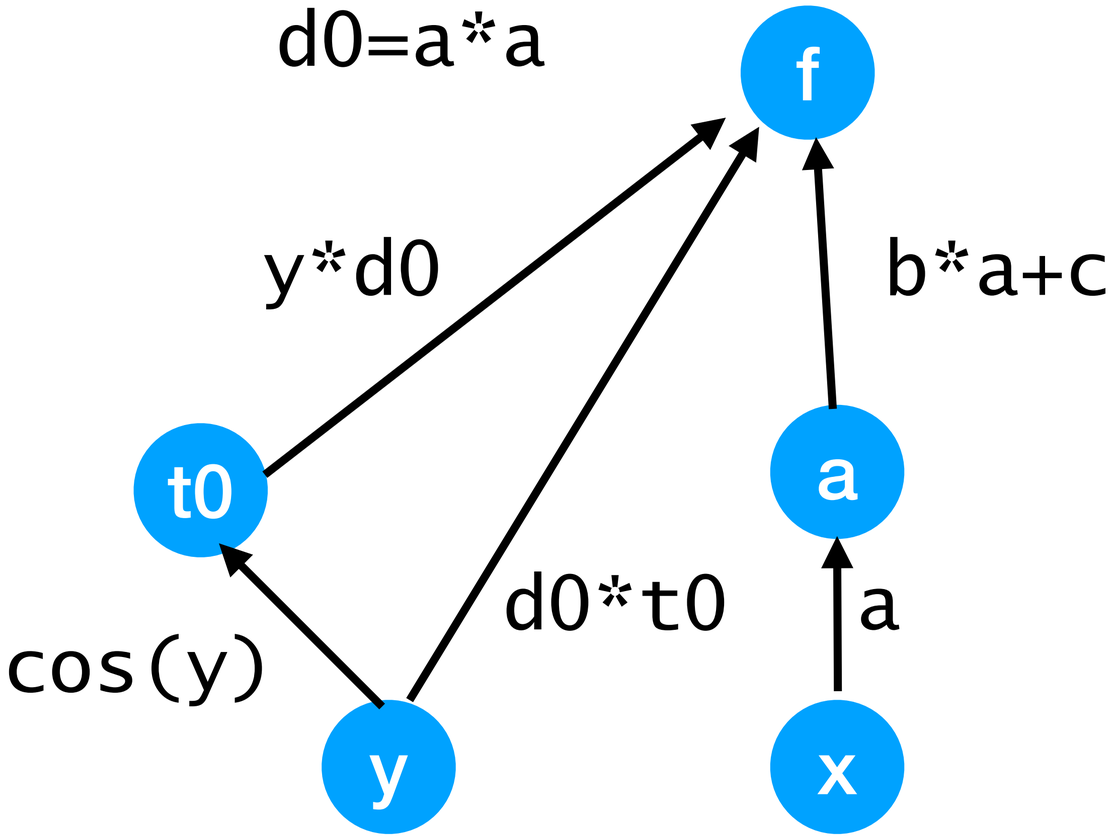
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

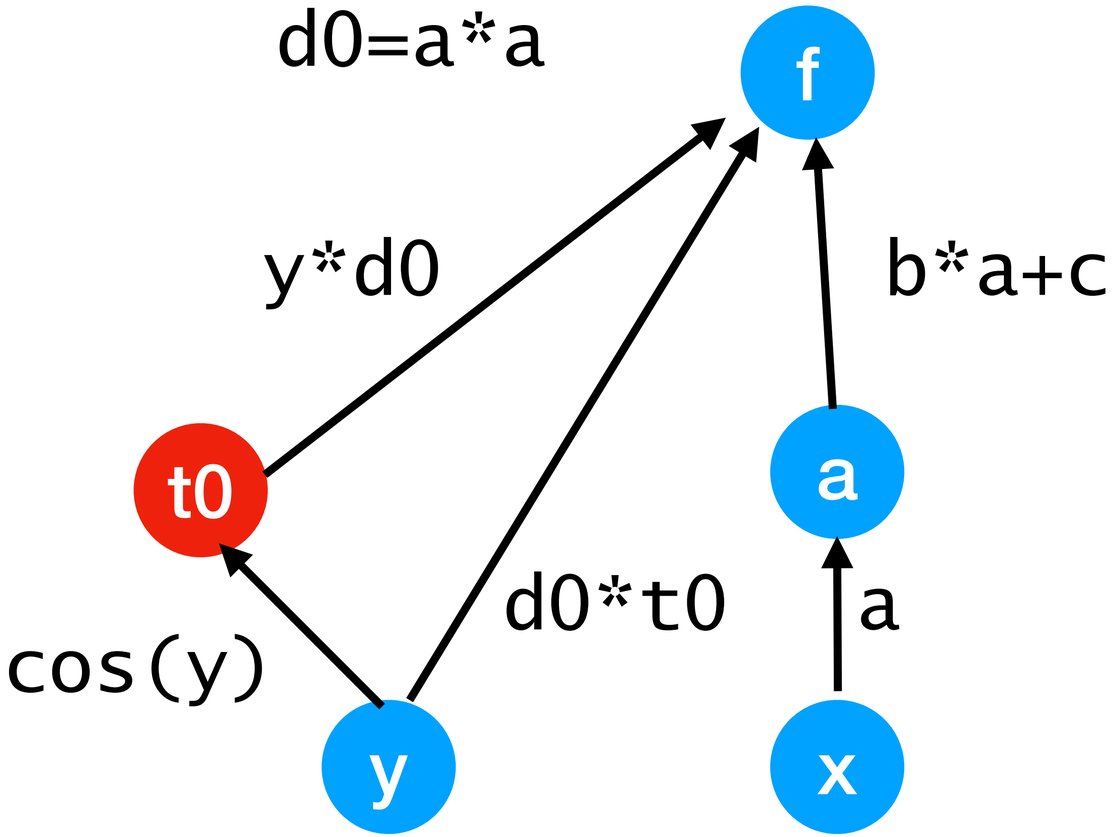
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

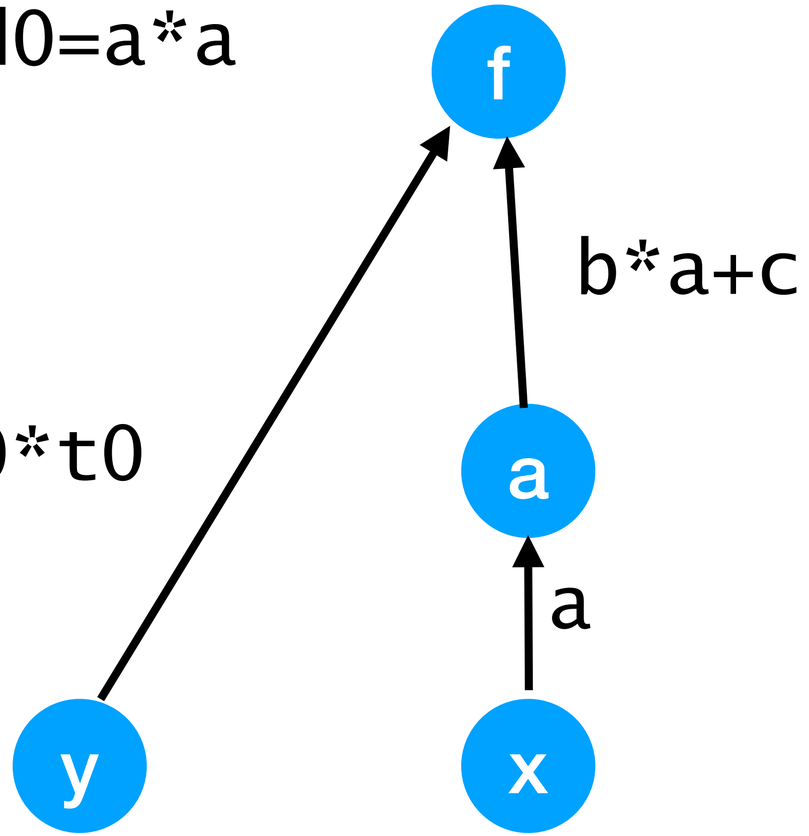


example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

d0=a*a



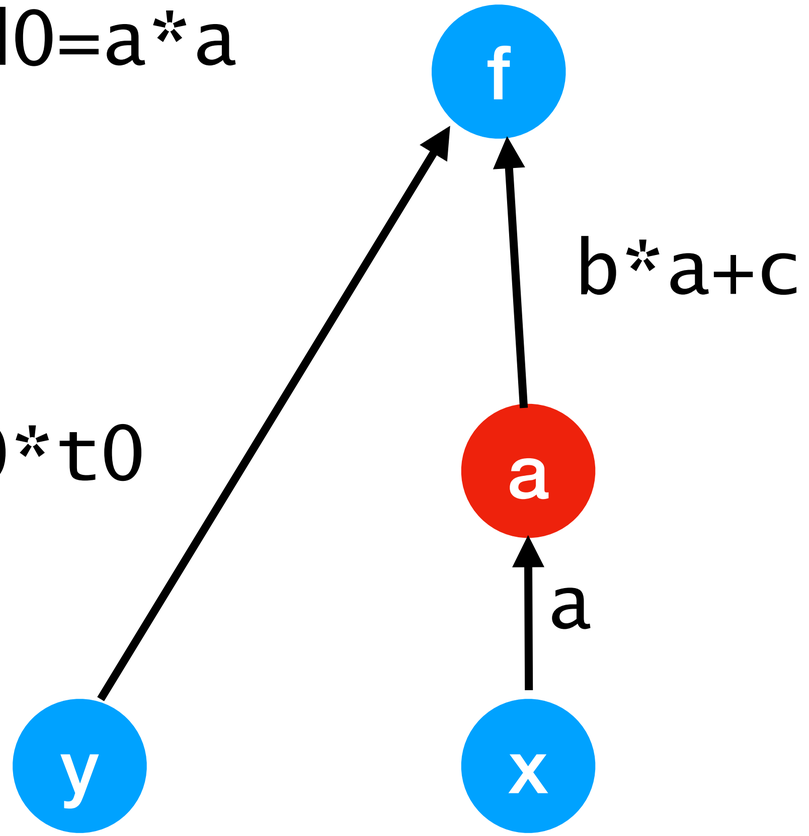
cos(y)*y*d0+d0*t0

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

d0=a*a

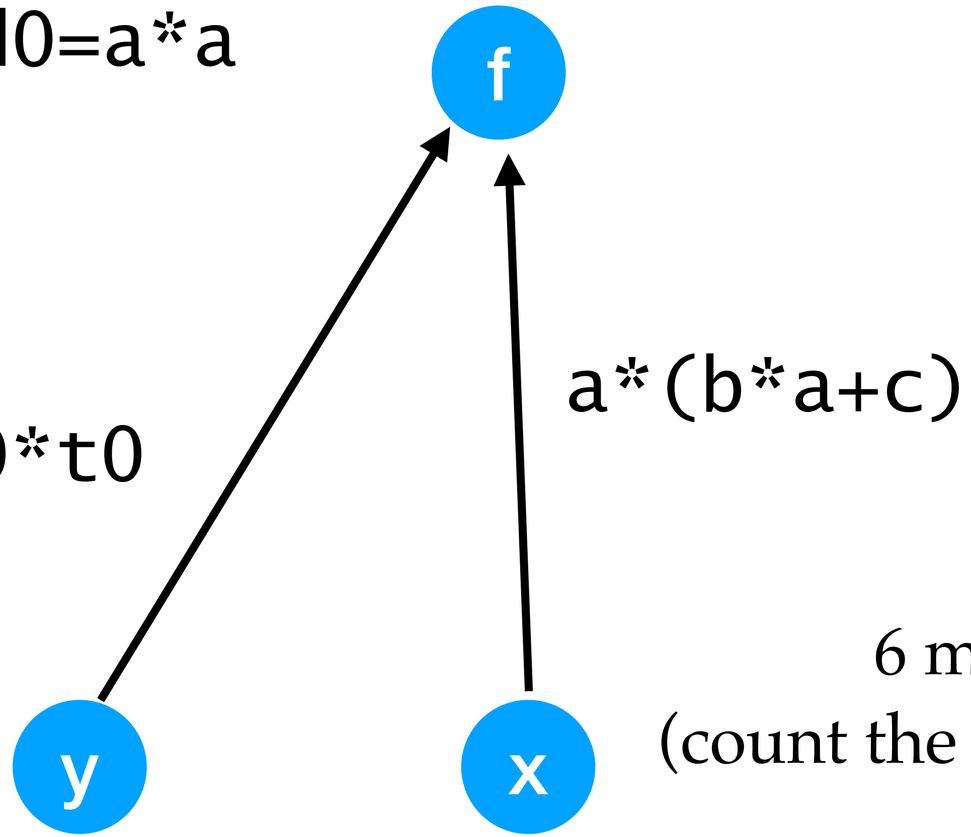


cos(y)*y*d0+d0*t0

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



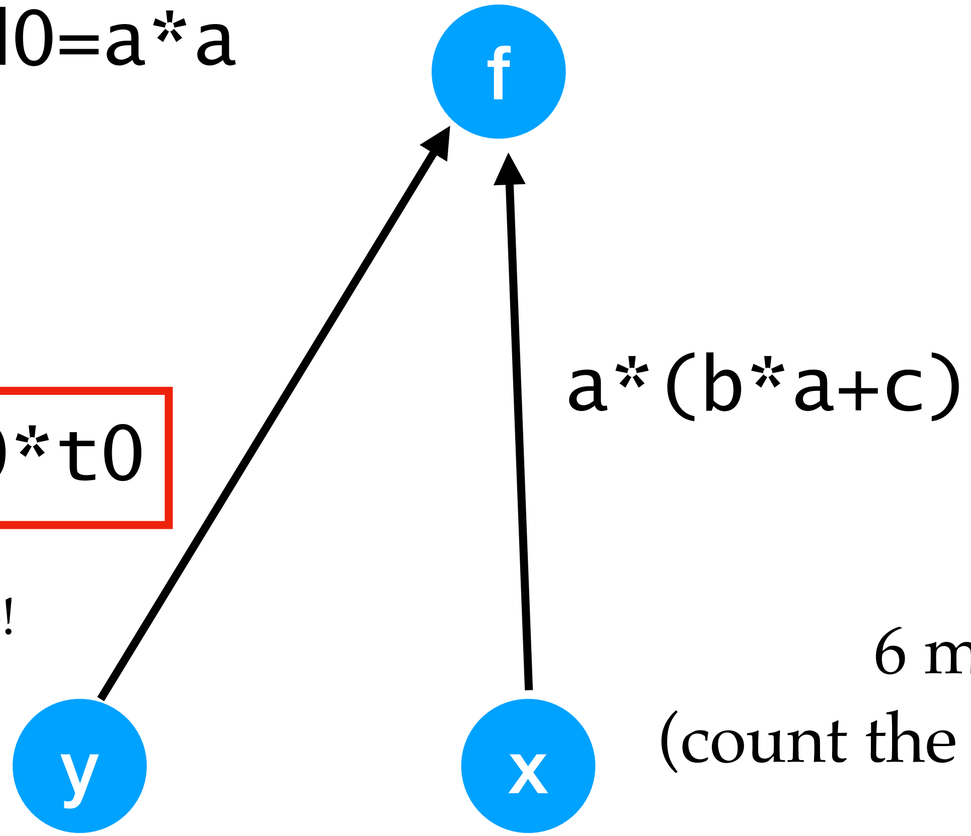
6 mults, 2 adds
(count the number of * and +)

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Reverse mode: eliminate vertices from outputs to inputs

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```

d0=a*a



a*(b*a+c)

cos(y)*y*d0+d0*t0

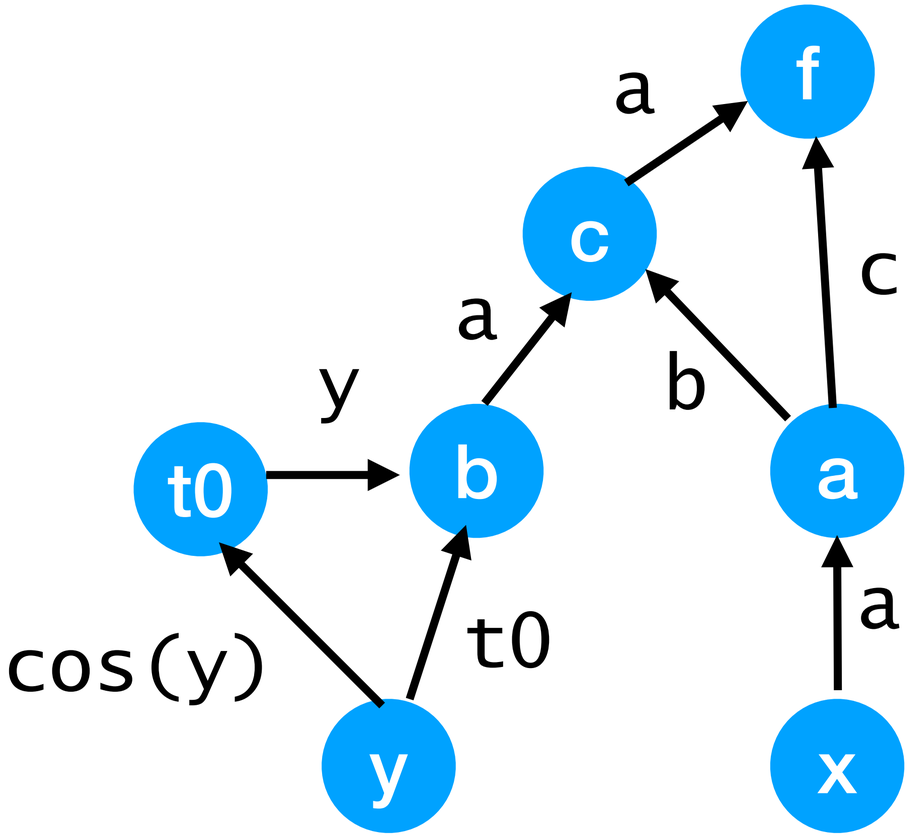
reverse mode fails to factor out **d0** here!

6 mults, 2 adds
(count the number of * and +)

example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

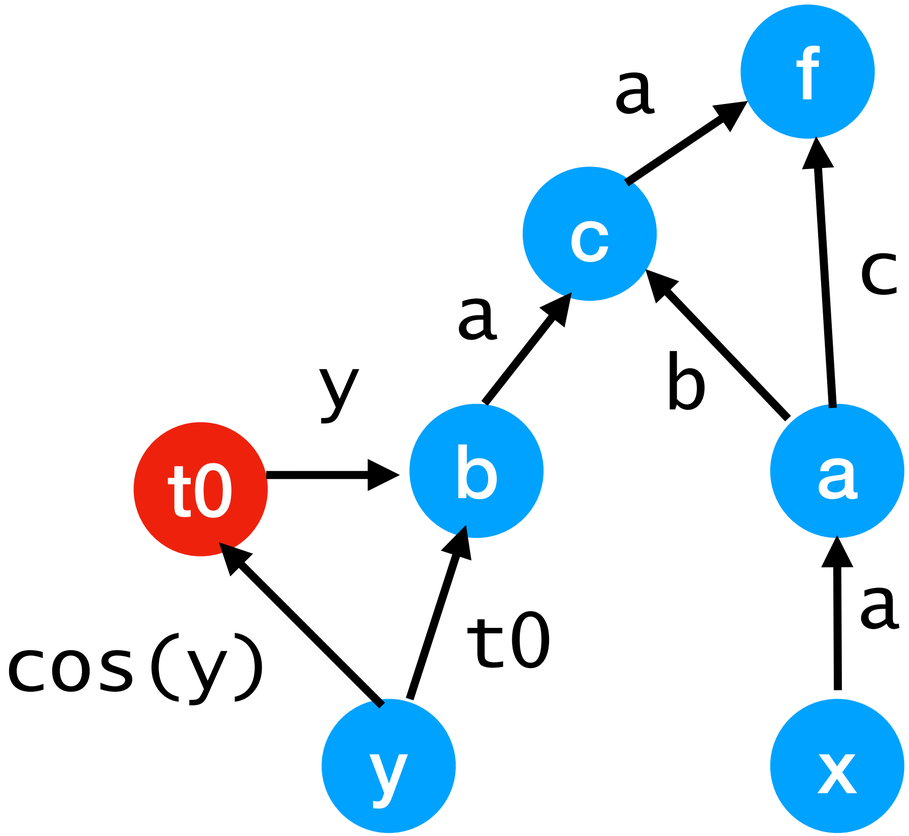
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

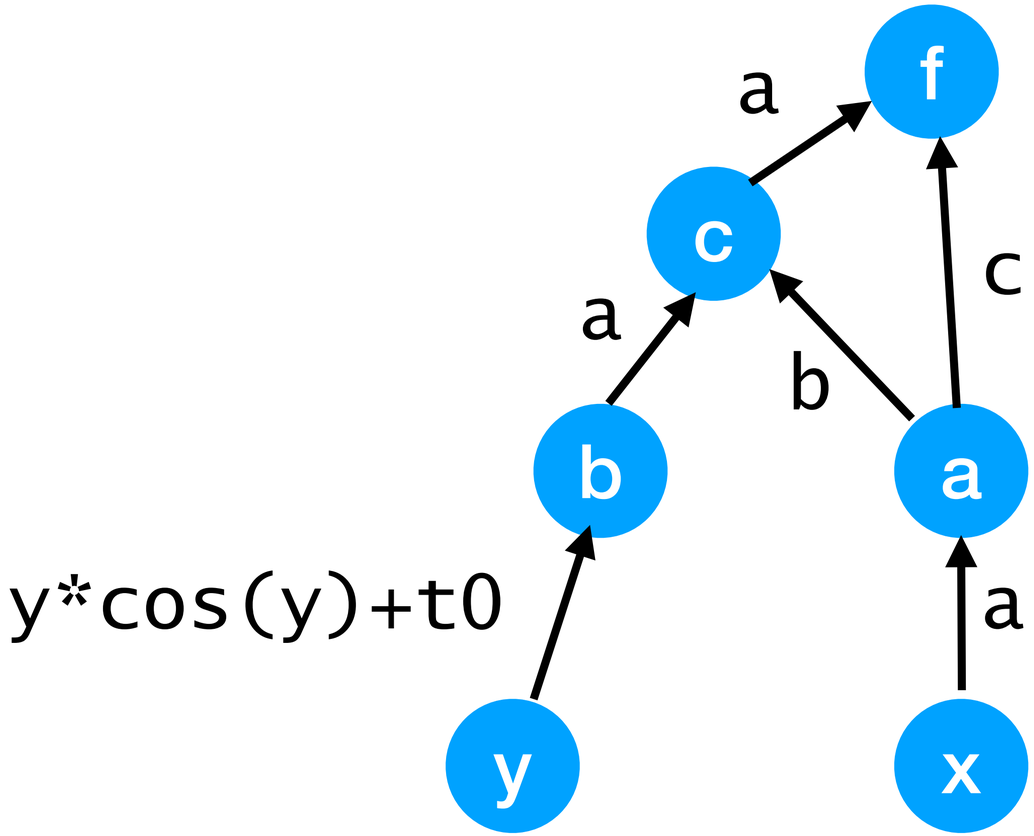
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

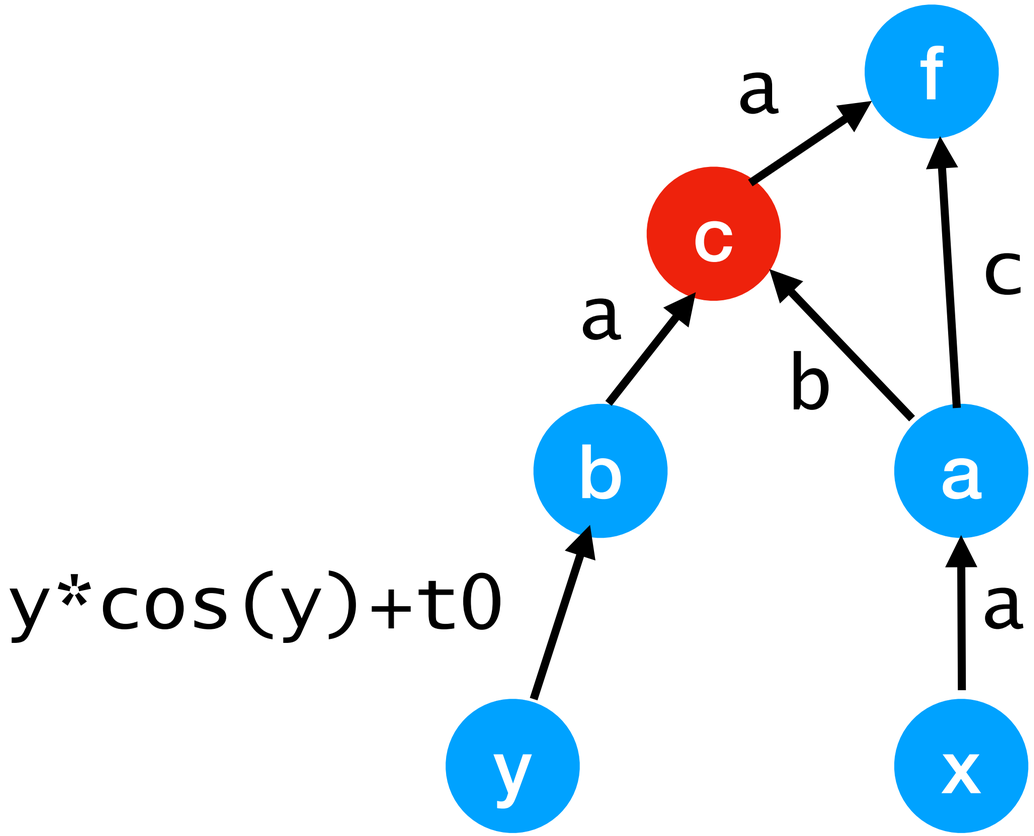
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

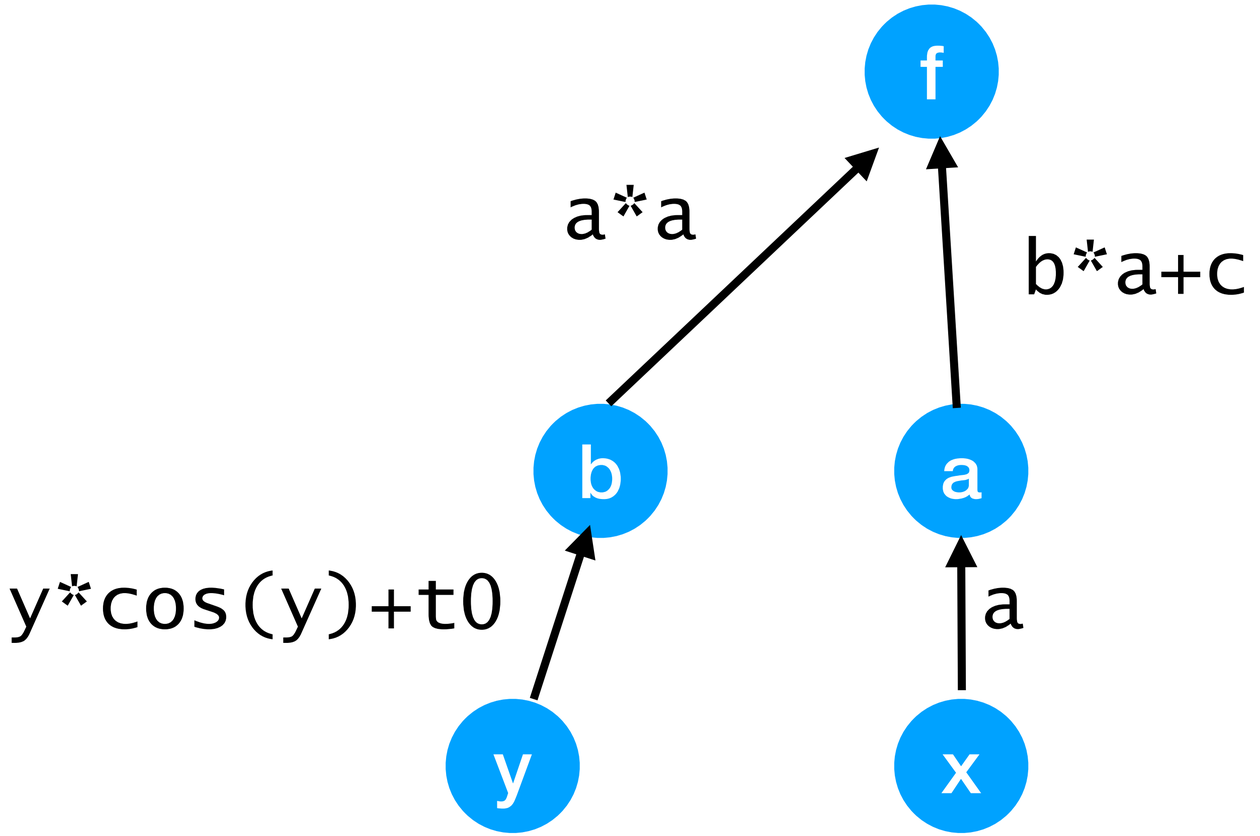
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

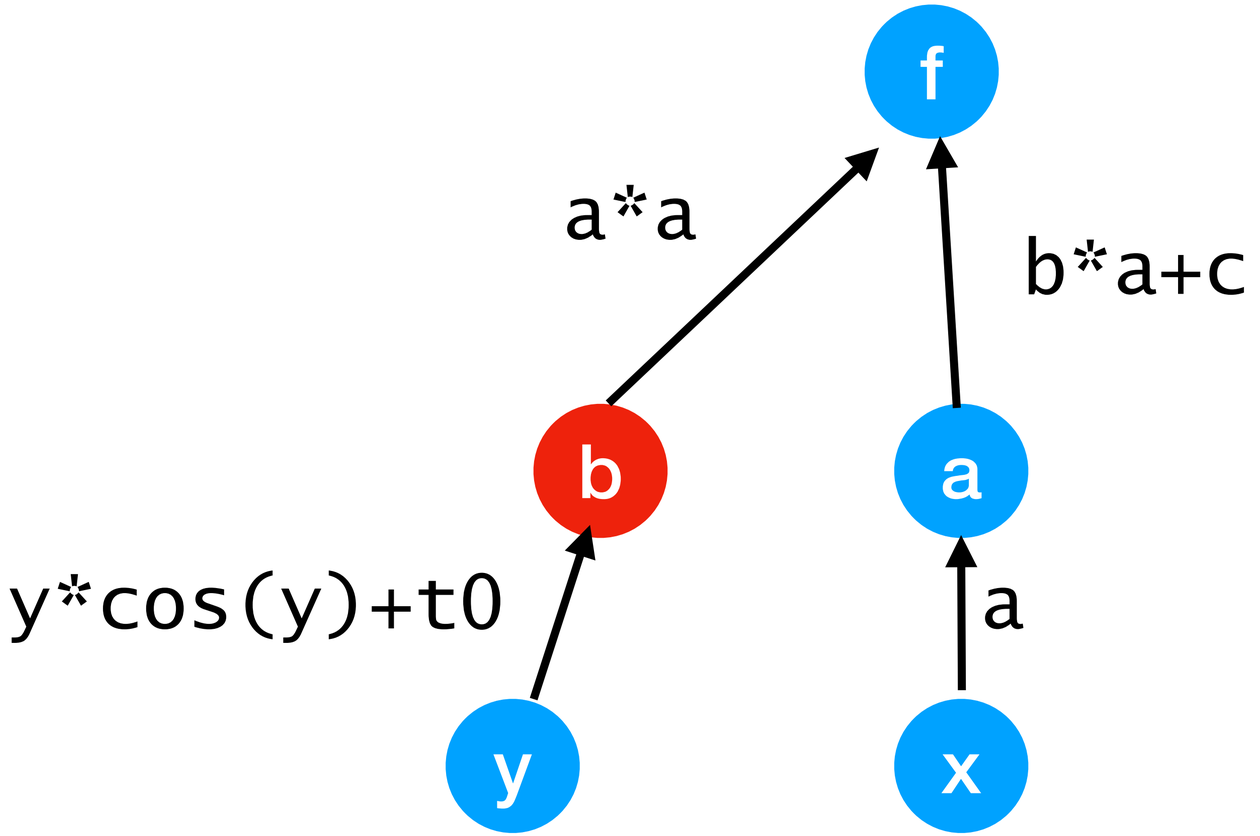
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

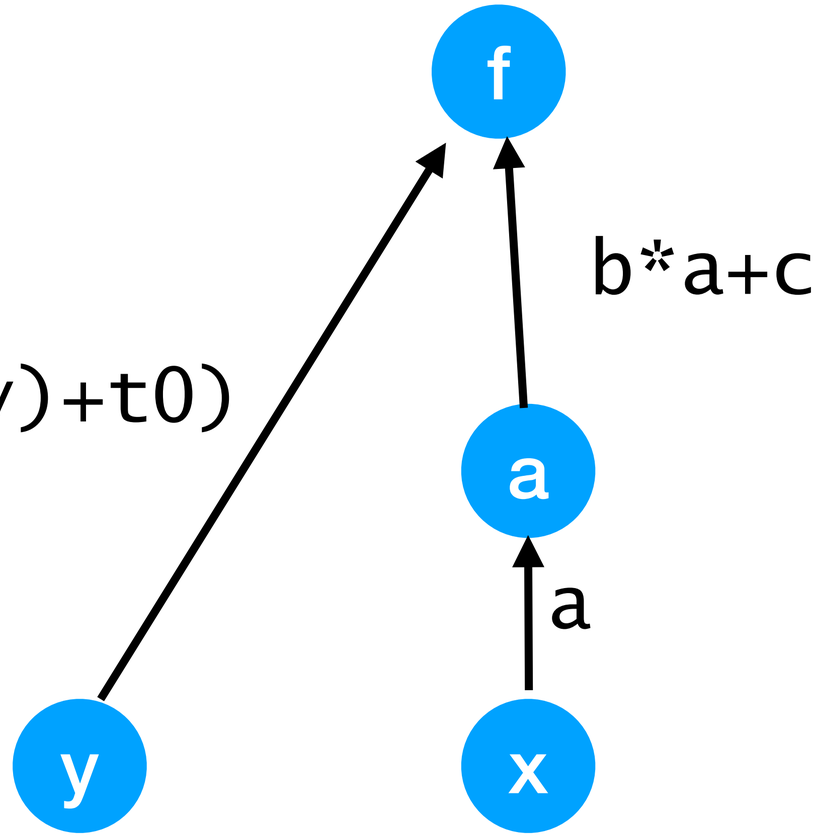
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

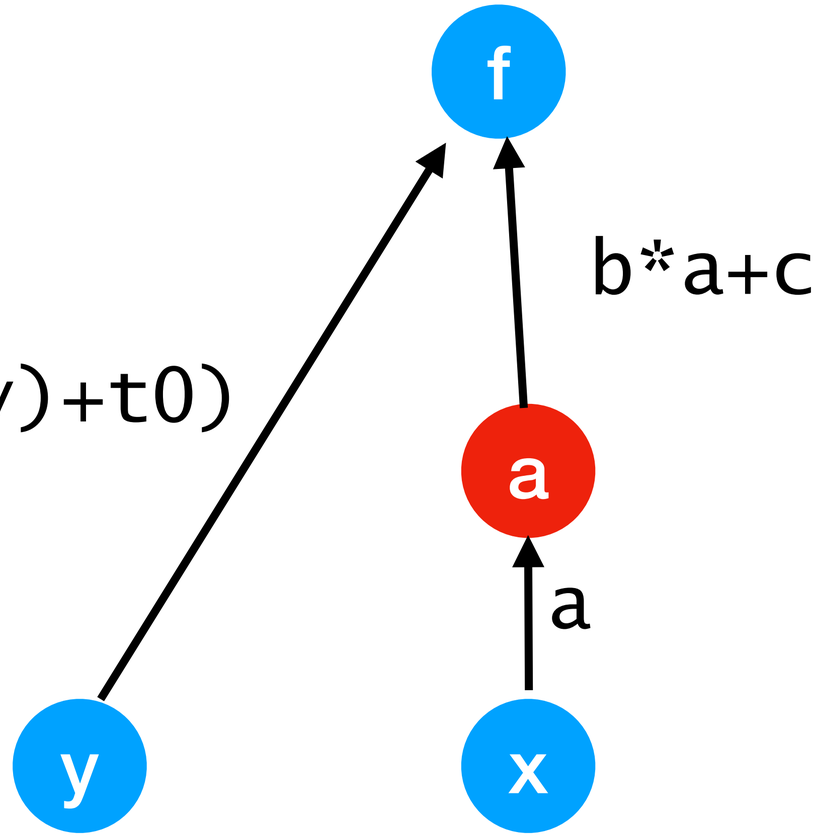
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

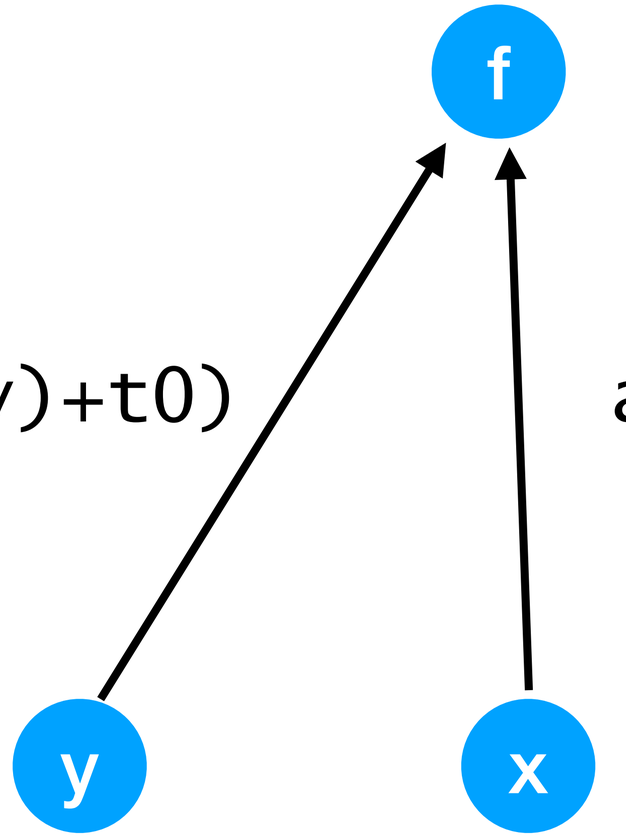
```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Cross-country mode: eliminate vertices in some chosen order

```
t0 = sin(y)
b = t0 * y
a = exp(x)
c = a * b
f = a * c
```



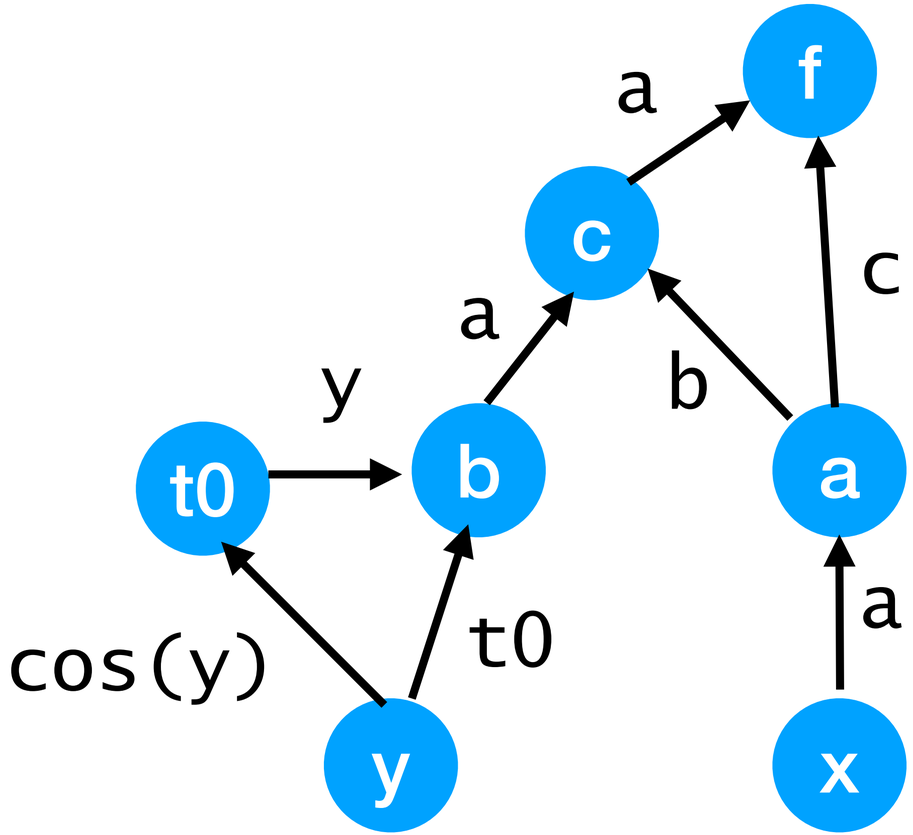
a*a*(y*cos(y)+t0)

a*(b*a+c)

5 mults, 2 adds
(count the number of * and +)
probably optimal in this case

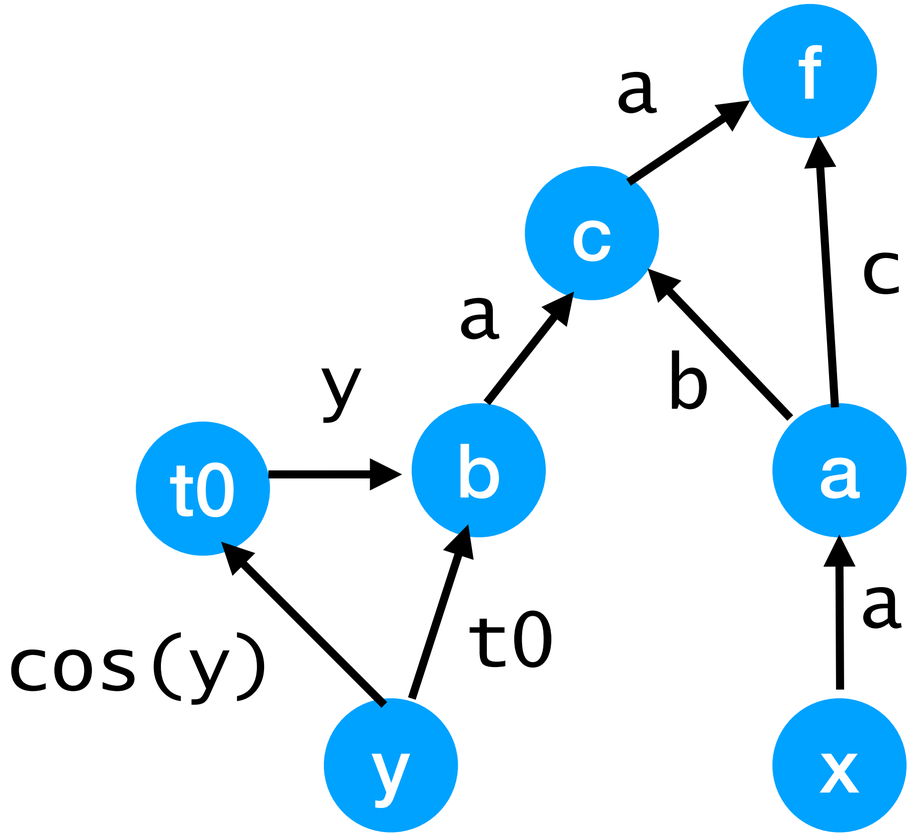
example taken from
https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

The optimal Jacobian accumulation problem



given a computational graph of primal code,
find the Jacobian computation that results in minimal mults and adds

The optimal Jacobian accumulation problem



given a computational graph of primal code,
find the Jacobian computation that results in minimal mults and adds

Math. Program., Ser. A (2008) 112:427–441
DOI 10.1007/s10107-006-0042-z

FULL LENGTH PAPER

Optimal Jacobian accumulation is NP-complete

Uwe Naumann

in fact, even optimal vertex elimination doesn't necessarily achieve optimal result!

Some trivia

- for scalar output, reverse mode is within 2x of optimal
- eliminating one edge at a time (edge elimination) can give better results
- eliminating two edges at a time (face elimination) can give even better result

https://cscapes.cs.purdue.edu/pub/hovland_cscapes2008-adintro.pdf

Determining vertex / subgraph elimination order

- proper way: integer programming

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

$$\text{subject to} \quad \sum_{i \in V} x_{it} = 1 \quad \forall\, t \in T \qquad (2)$$

$$\sum_{t \in T} x_{it} = 1 \quad \forall\, i \in Z \qquad (3)$$

$$x_{it} = 0 \quad \forall\, i \in X \cup Y,\ \forall\, t \in T \qquad (4)$$

$$\left.\begin{aligned} d_{ijt} &\geq x_{it} + c_{ij(t-1)} - 1 \\ d_{ijt} &\geq x_{jt} + c_{ij(t-1)} - 1 \\ d_{ijt} &\leq x_{it} + x_{jt} \\ d_{ijt} &\leq c_{ij(t-1)} \end{aligned}\right\} \quad \forall\, (i,j) \in \tilde{E},\ \forall\, t \in T \qquad (5)$$

$$f_{ijt} \geq d_{ikt} + d_{kjt} - 1 \quad \forall\, (i,j) \in \tilde{E},\ \forall\, k \in Z,\ \forall\, t \in T \qquad (6)$$

$$\left.\begin{aligned} c_{ijt} &\geq f_{ijt} \\ c_{ijt} &\leq 1 - d_{ijt} \\ c_{ijt} &\leq c_{ij(t-1)} + (f_{ijt} + d_{ijt}) \\ c_{ijt} &\geq c_{ij(t-1)} - (f_{ijt} + d_{ijt}) \end{aligned}\right\} \quad \forall\, (i,j) \in \tilde{E},\ \forall\, t \in T \qquad (7)$$

$$x_{it} \in \{0,1\} \quad \forall\, i \in V,\ \forall\, t \in T \qquad (8)$$

$$c_{ijt},\ d_{ijt},\ f_{ijt} \in \{0,1\} \quad \forall\, (i,j) \in \tilde{E},\ \forall\, t \in T \qquad (9)$$

An Integer Programming Approach to Optimal Derivative Accumulation

Jieqiu Chen, Paul Hovland, Todd Munson, and Jean Utke

Determining vertex/subgraph elimination order

- heuristics
  - greedy Markowitz (remove vertex where input*output degree is the smallest)
  - D* (removing a subgraph that appears the most times in the computation)
    see my implementation https://github.com/BachiLi/cdstar — no guarantee on correctness!!

Efficient Symbolic Differentiation for Graphics Applications

Brian Guenter
Microsoft Research

Optimal Jacobian accumulation is related to the "multivariate Horner scheme"

an efficient way to evaluate multivariate polynomials (also NP-complete, obviously)

On the multivariate Horner scheme

J. M. Peña and Thomas Sauer

Abstract. We present an efficient version of the Horner scheme for the evaluation of multivariate polynomials and study its stability properties.

Greedy Algorithms for Optimizing Multivariate Horner Schemes

Martine Ceberio and Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA,
{mceberio,vladik}@cs.utep.edu

Abstract

For univariate polynomials $f(x_1)$, Horner's scheme provides the fastest way to compute a value. For multivariate polynomials, several different version of Horner's scheme are possible; it is not clear which of them is optimal. In this paper, we propose a greedy algorithm, which it is hoped will lead to good computation times.

Optimal Jacobian accumulation is related to tensor network evaluation

aka “einsum”

$$M_{pqrs} = C_{pi} C_{qj} I_{ijkl} C_{rk} C_{sl}$$

PATH INFORMATION:

- Introduction
- The Optimal Path
- The Branching Path
- The Greedy Path
- The Random-Greedy Path
- The Dynamic Programming Path
- Custom Path Optimizers

Hyper-optimized tensor network contraction

Johnnie Gray[1,2] and Stefanos Kourtis[1,3,4]

[1]Blackett Laboratory, Imperial College London, London SW7 2AZ, United Kingdom
[2]Div
[3]Dep
[4]Inst
Ma

Algorithms for Tensor Network Contraction Ordering

Frank Schindler[1,2] and Adam S. Jermyn[3,2,*]

[1] *Department of Physics, University of Zurich, Winterthurerstrasse 190, 8057 Zurich, Switzerland*
[2] *Kavli Institute for Theoretical Physics, University of California, Santa Barbara, CA 93106, USA*
[3] *Center for Computational Astrophysics, Flatiron Institute, New York, NY 10010, USA*

(Dated: January 23, 2020)

https://optimized-einsum.readthedocs.io/en/stable/

https://docs.juliahub.com/ITensors/P3pqL/0.2.3/ContractionSequenceOptimization.html

When do we need the full Jacobian?

When do we need the full Jacobian?

Gauss–Newton algorithm

Article Talk

From Wikipedia, the free encyclopedia

The **Gauss–Newton algorithm** is used to solve non-linear least squares problems, which is equivalent to minimizing a sum of squared function values. It is an extension of Newton's method for finding a minimum of a non-linear function. Since a sum of squares must be nonnegative, the algorithm can be viewed as using Newton's method to iteratively approximate zeroes of the components of the sum, and thus minimizing the sum. In this sense, the algorithm is also an effective method for solving overdetermined systems of equations. It has the advantage that second derivatives, which can be challenging to compute, are not required.[1]

Non-linear least squares problems arise, for instance, in non-linear regression, where parameters in a model are sought such that the model is in good agreement with available observations.

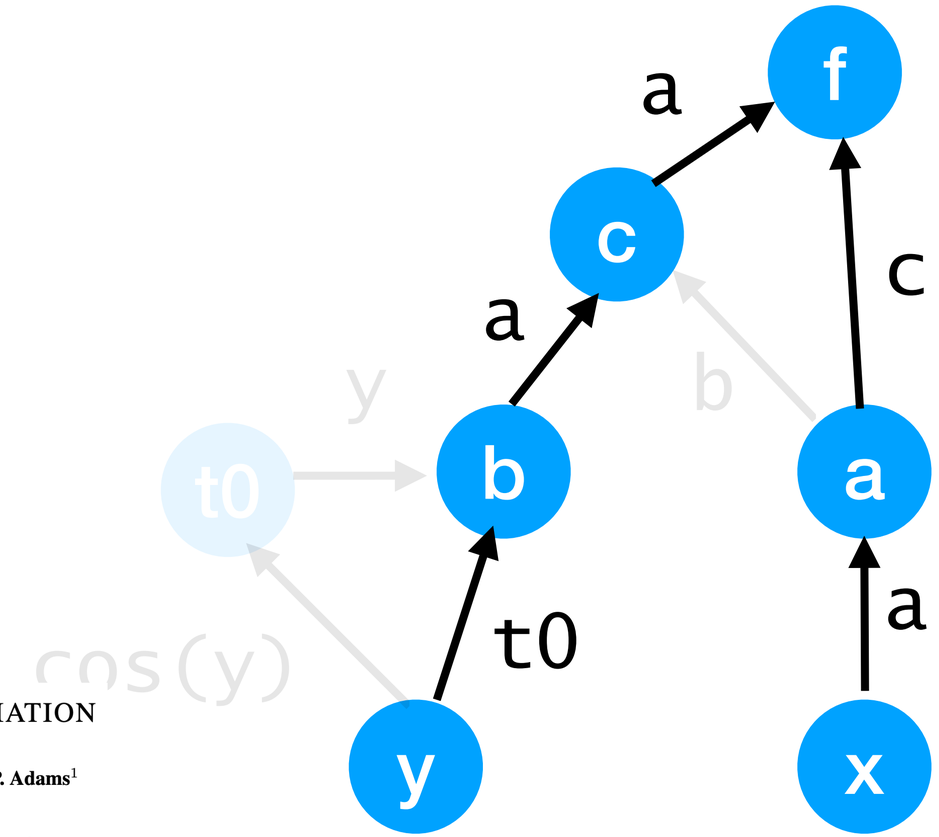
The method is named after the mathematicians C
appeared in Gauss' 1809 work *Theoria motus co*
ambientum.[2]

At each iteration, the update $\Delta = \boldsymbol{\beta}^{(s+1)} - \boldsymbol{\beta}^{(s)}$ can be found by rearranging the previous equation in the following two steps:

- $\Delta = -\left(\mathbf{J_r}^{\mathrm{T}}\mathbf{J_r}\right)^{-1}\mathbf{J_r}^{\mathrm{T}}\mathbf{r}\left(\boldsymbol{\beta}^{(s)}\right)$
- $\mathbf{J_r}^{\mathrm{T}}\mathbf{J_r}\Delta = -\mathbf{J_r}^{\mathrm{T}}\mathbf{r}\left(\boldsymbol{\beta}^{(s)}\right)$

Randomized AD: an interesting alternative

idea: when traversing the graph, choose a random subset of the paths



RANDOMIZED AUTOMATIC DIFFERENTIATION

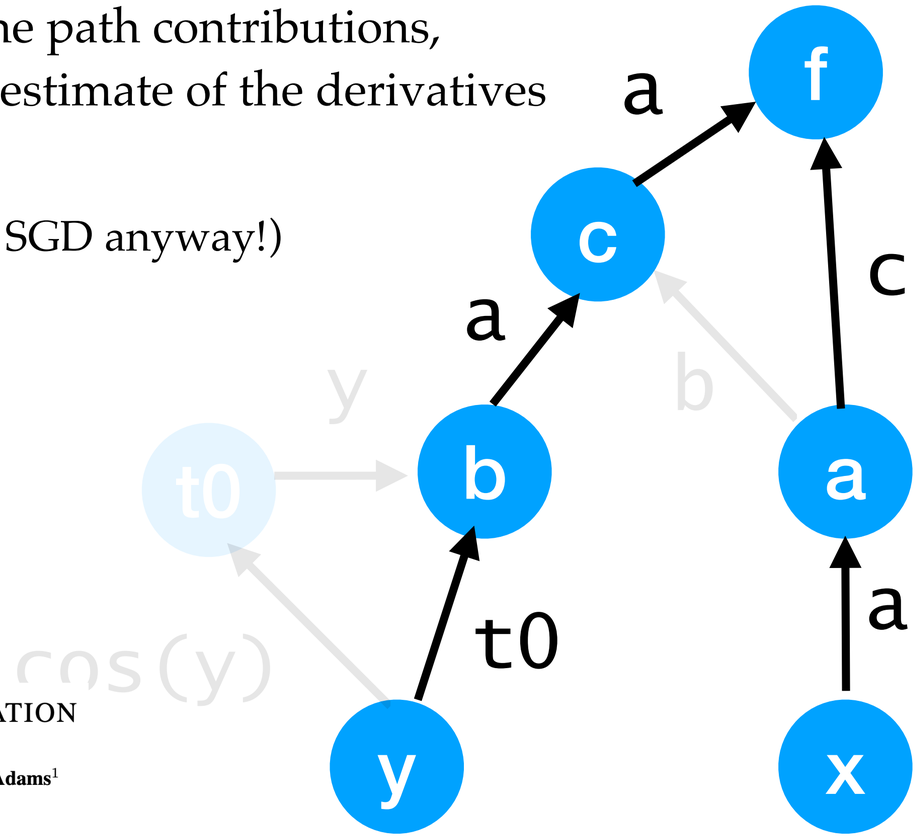
Deniz Oktay[1], **Nick McGreivy**[2], **Joshua Aduol**[1], **Alex Beatson**[1], **Ryan P. Adams**[1]
Princeton University
Princeton, NJ
{doktay,mcgreivy,jaduol,abeatson,rpa}@princeton.edu

Randomized AD: an interesting alternative

idea: when traversing the graph, choose a random subset of the paths

by properly weighting the path contributions,
we can compute an unbiased estimate of the derivatives

(we use noisy gradients for SGD anyway!)



$$E[\,\cdot\,] = \frac{\partial f}{\partial x}$$

$$E[\,\cdot\,] = \frac{\partial f}{\partial y}$$

RANDOMIZED AUTOMATIC DIFFERENTIATION

Deniz Oktay[1], **Nick McGreivy**[2], **Joshua Aduol**[1], **Alex Beatson**[1], **Ryan P. Adams**[1]
Princeton University
Princeton, NJ
{doktay,mcgreivy,jaduol,abeatson,rpa}@princeton.edu

Another kind of randomized AD: randomized telescoping sum

primal code:

```
// n is super large
for i in range(n):
   …
```

reverse mode:

```
m = sample()
for i in range(m):
   …

for i in reversed(range(m)):
   …
```

with m properly sampled and gradients properly weighted, the gradients can be made unbiased!

Efficient Optimization of Loops and Limits with Randomized Telescoping Sums

---

Alex Beatson [1] **Ryan P. Adams** [1]

Lots of open problems

- how do we apply cross country mode to control flow?
- can we provide worst case bounds for these heuristics?
- is there a better cost function? (counting mults and adds is a bad cost for modern machines)
- how to take parallelism into account? checkpointing?
- can AlphaGo solve the Jacobian accumulation problem?