Fun with Semirings

A functional pearl on the abuse of linear algebra

Stephen Dolan

Computer Laboratory
University of Cambridge
stephen.dolan@cl.cam.ac.uk

September 25, 2013

Linear algebra is magic

If your problem can be expressed as vectors and matrices, it is essentially already solved.

Linear algebra works with *fields*, like the real or complex numbers: sets with a notion of addition, multiplication, subtraction and division.

We don't have fields

CS has many structures with "multiplication" and "addition":

- conjunction and disjunction
- sequencing and choice
- intersection and union
- product type and sum type

But very few with a sensible "division" or "subtraction".

What we have are *semirings*, not fields.

Semirings

A *closed semiring* is a set with some notion of addition and multiplication as well as a unary operation *, where:

$a + b = b + a$	$(+, 0)$ is a commutative monoid
$a + (b + c) = (a + b) + c$	
$a + 0 = a$	
$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	$(\cdot, 1)$ is a monoid, with zero
$a \cdot 1 = 1 \cdot a = a$	
$a \cdot 0 = 0 \cdot a = 0$	
$a \cdot (b + c) = a \cdot b + a \cdot c$	$\cdot$ distributes over $+$
$(a + b) \cdot c = a \cdot c + b \cdot c$	
$a^* = 1 + a \cdot a^*$	closure operation

Daniel J. Lehmann, Algebraic Structures for Transitive Closure, 1977.

Closed semirings

A closed semiring has a closure operation *, where

$$a^* = 1 + a \cdot a^* = 1 + a^* \cdot a$$

Intuitively, we can often think of closure as:

$$a^* = 1 + a + a^2 + a^3 + \dots$$

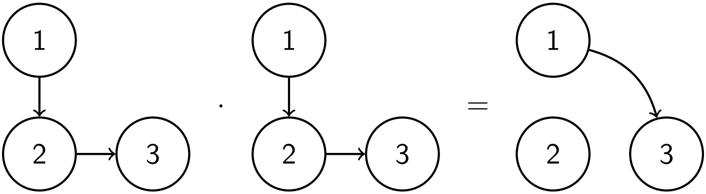
Closed semirings as a Haskell typeclass

```haskell
infixl 9 @.
infixl 8 @+
class Semiring r where
  zero, one :: r
  closure :: r -> r
  (@+), (@.) :: r -> r -> r
```

```haskell
instance Semiring Bool where
  zero = False
  one = True
  closure x = True
  (@+) = (||)
  (@.) = (&&)
```

Adjacency matrices

Directed graphs are represented as matrices of Booleans.
G^2 gives the two-hop paths through G.

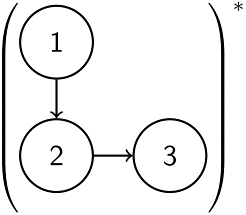


$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

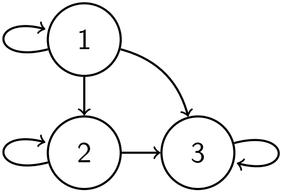
$$\begin{aligned} (AB)_{ij} &= \sum_k A_{ik} \cdot B_{kj} \\ &= \exists k \text{ such that } A_{ik} \wedge B_{kj} \end{aligned}$$

Closure of an adjacency matrix

The closure of an adjacency matrix gives us the reflexive transitive closure of the graph.



=



$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} A^* &= 1 + A \cdot A^* \\ &= 1 + A + A^2 + A^3 + \dots \end{aligned}$$

A semiring of matrices

A matrix is represented by a list of lists of elements.

```
data Matrix a = Matrix [[a]]
instance Semiring a => Semiring (Matrix a) where
    ...
```

Matrix addition and multiplication is as normal, and Lehmann gives an imperative algorithm for calculating the closure of a matrix.

Closure of a matrix

The correctness proof of the closure algorithm states:

$$\text{If } M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

$$\text{then } M^* = \begin{pmatrix} A^* + B' \cdot \Delta^* \cdot C' & B' \cdot \Delta^* \\ \Delta^* \cdot C' & \Delta^* \end{pmatrix}$$

where $B' = A^* \cdot B$, $C' = C \cdot A^*$ and $\Delta = D + C \cdot A^* \cdot B$.

Block matrices

We can split a matrix into blocks, and join them back together.

```
type BlockMatrix a = (Matrix a, Matrix a,
                      Matrix a, Matrix a)

msplit :: Matrix a -> BlockMatrix a
mjoin :: BlockMatrix a -> Matrix a
```

Closure of a matrix

The algorithm is imperative, but the *correctness proof* gives a recursive functional implementation:

```
closure (Matrix [[x]]) = Matrix [[closure x]]
closure m = mjoin
  (first' @+ top' @. rest' @. left', top' @. rest',
   rest' @. left',                   rest')
  where
    (first, top, left, rest) = msplit m
    first' = closure first
    top' = first' @. top
    left' = left @. first'
    rest' = closure (rest @+ left' @. top)
```

Shortest distances in a graph

Distances form a semiring, with $\cdot$ as addition and $+$ as choosing the shorter. The closure algorithm then finds shortest distances.

```
data ShortestDistance = Distance Int | Unreachable
instance Semiring ShortestDistance where
  zero = Unreachable
  one = Distance 0
  closure x = one

  x @+ Unreachable = x
  Unreachable @+ x = x
  Distance a @+ Distance b = Distance (min a b)

  x @. Unreachable = Unreachable
  Unreachable @. x = Unreachable
  Distance a @. Distance b = Distance (a + b)
```

Shortest paths in a graph

We can also recover the actual path:

```
data ShortestPath n = Path Int [(n,n)] | NoPath
instance Ord n => Semiring (ShortestPath n) where
  zero = NoPath
  one = Path 0 []
  closure x = one

  x @+ NoPath = x
  NoPath @+ x = x
  Path a p @+ Path a' p'
    | a < a'              = Path a p
    | a == a' && p < p'   = Path a p
    | otherwise           = Path a' p'

  x @. NoPath = NoPath
  NoPath @. x = NoPath
  Path a p @. Path a' p' = Path (a + a') (p ++ p')
```

Solving linear equations

If we have a linear equation like:

$$x = a \cdot x + b$$

then $a^* \cdot b$ is a solution:

$$\begin{aligned} a^* \cdot b &= (a \cdot a^* + 1) \cdot b \\ &= a \cdot (a^* \cdot b) + b \end{aligned}$$

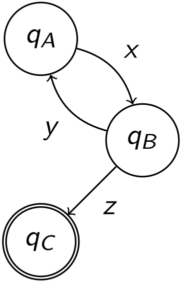
If we have a system of linear equations like:

$$\begin{aligned} x_1 &= A_{11} x_1 + A_{12} x_2 + \ldots A_{1n} x_n + B_1 \\ &\vdots \\ x_n &= A_{n1} x_1 + A_{n2} x_2 + \ldots A_{nn} x_n + B_n \end{aligned}$$

then $A^* \cdot B$ is a solution (for a matrix A and vector B of coefficients) which can be found using `closure`.

Regular expressions and state machines

A state machine can be described by a regular grammar:



$$
\begin{aligned}
A &\to xB \\
B &\to yA + zC \\
C &\to 1
\end{aligned}
$$

The regular grammar is a system of linear equations, and the regular expression describing it can be found by `closure`.

Solving linear equations

Reconstructing regular expressions
Solving equations in the "free" semiring rebuilds regular expressions from a state machine.

Dataflow analysis
Solving equations in the semiring of sets of variables does live variables analysis (among others).

Linear recurrences and power series

Suppose the next value in a sequence is a linear combination of previous values:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-2) + F(n-1)
\end{aligned}
$$

We represent these as *formal power series*:

$$F = x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 \dots$$

Multiplying by x shifts the sequence one place, so:

$$F = 1 + (x^2 + x) \cdot F$$

Power series are a semiring

We represent power series as lists: `a::p` is $a + px$.

```
instance Semiring r => Semiring [r] where
  zero = []
  one = [one]
```

Addition is pointwise:

```
  [] @+ y = y
  x @+ [] = x
  (x:xs) @+ (y:ys) = (x @+ y):(xs @+ ys)
```

Multiplying power series

Multiplying power series works like this:

$$(a + px)(b + qx) = ab + (aq + pb + pqx)x$$

In Haskell:

```
[] @. _ = []
_ @. [] = []
(a:p) @. (b:q) = (a @. b):(map (a @.) q @+
                            map (@. b) p @+
                            (zero:(p @. q)))
```

This is convolution, without needing indices.

M. Douglas McIlroy. Power series, power serious. Journal of Functional Programming, 1999.

Closure of a power series

The closure of $a + px$ must satisfy:

$$(a + px)^* = 1 + (a + px)^* \cdot (a + px)$$

This has a solution satisfying:

$$(a + px)^* = a^* \cdot (1 + px \cdot (a + px)^*)$$

which translates neatly into (lazy!) Haskell:

```
closure [] = one
closure (a:p) = r
  where r = [closure a] @. (one:(p @. r))
```

Fibonacci, again

$$
\begin{aligned}
F &= 1 + (x + x^2)F \\
&= (x + x^2)^*
\end{aligned}
$$

```
fib = closure [0,1,1]
```

Any linear recurrence can be solved with `closure`.

Knapsacks

Suppose we are trying to fill our baggage allowance with:

Knuth books:	weight 10, value 100
Haskell books:	weight 7, value 80
Java books:	weight 9, value 3

The best value we can have with weight n is:

$$\text{best}_n = \max(100 + \text{best}_{n-10}, 80 + \text{best}_{n-7}, 3 + \text{best}_{n-9})$$

In the $(\max, +)$-semiring, that reads:

$$\text{best}_n = 100 \cdot \text{best}_{n-10} + 80 \cdot \text{best}_{n-7} + 3 \cdot \text{best}_{n-9}$$

which is a linear recurrence.

Thank you!

Questions?

Many problems are linear, for a suitable notion of "linear".

`stephen.dolan@cl.cam.ac.uk`

Live variables analysis

Many dataflow analyses are just linear equations in a semiring.
This live variables analysis uses the semiring of sets of variables.

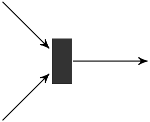
```
A  x := 1

B  while x < y:

C      x := x * 2

D  return x
```

$$\mathrm{IN}_A = \mathrm{OUT}_A \cap \overline{\{\mathtt{x}\}}$$
$$\mathrm{IN}_B = \mathrm{OUT}_B \cup \{\mathtt{x}, \mathtt{y}\}$$
$$\mathrm{IN}_C = \mathrm{OUT}_C \cup \{\mathtt{x}\}$$
$$\mathrm{IN}_D = \mathrm{OUT}_D \cup \{\mathtt{x}\}$$

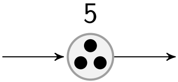
$$\mathrm{OUT}_A = \mathrm{IN}_B$$
$$\mathrm{OUT}_B = \mathrm{IN}_C \cup \mathrm{IN}_D$$
$$\mathrm{OUT}_C = \mathrm{IN}_B$$
$$\mathrm{OUT}_D = \emptyset$$

Petri nets

Timed event graphs (a form of Petri net with a notion of time) can be viewed as "linear" systems, in the $(\max, +)$-semiring



This transition fires for the nth time after all of its inputs have fired for the nth time.



The nth token is available from this place 5 time units after then $(n-3)$th token is available from its input.

G. Cohen, P. Moller, J.P. Quadrat, M. Viot, Linear system theory for discrete event systems, 1984.