Towards a general algorithm for path-finding problems

Juan Carlos Saenz-Carrasco

June 12, 2013

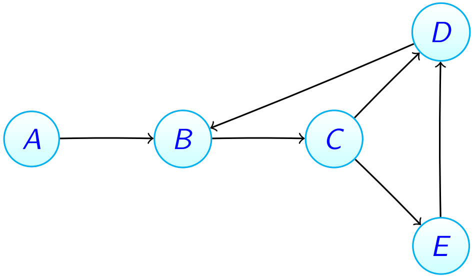
Generalities

Nowadays, path-finding problems are commonly found in:

- on the web,
- in traffic logistic,
- in operations research,
- in games,
- in string processing,
- and in many other contexts

What is a path?

In order to define a path, we first need a graph.
A graph is an ordered pair $G = < N, E >$, comprising a set of nodes (or vertices), in this case N, and a set of edges (or arcs) in this case E


$$\begin{tikzcd}
 & & & D \arrow[lll, bend left=0, shift left=0, ""] \\
A \arrow[r] & B \arrow[r] & C \arrow[ur] \arrow[dr] & \\
 & & & E \arrow[uu]
\end{tikzcd}$$

So, a path is . . .

A *path* in a graph can be defined in two ways:

- by giving a sequence of nodes,

  $$p = (n_0, n_1, \ldots, n_l), \text{ such that } l \geq 0$$

  and

  (n_i, n_{i+1}) is an edge of the graph, for $i = 0, 1, \ldots, l{-}1$

- or by giving an alternated sequence of nodes and edges

  $$q = (n_0, e_0, n_1, e_1, \ldots, e_{l-1}, n_l)$$

Paths and Edges

It is important to remark that a *path* $\neq$ *edge*, since the edge is always *one hop* between two nodes, whilst a path is comprised by *zero* (also called *empty path*) or more *hops*.
In other words, *paths* and *edges* have different *type*.

Mathematical Structures

A very significant feature about path-finding problems is that they can be modelled by using algebraic structures, specifically

- Matrices, in order to handle the graph
- and Semirings, in order to compute the operations over the matrices (as well other operations)

Let us start with Semirings

Definition (Semiring)

A *semiring* is a 5-tuple (A,1,0,$\otimes$,$\oplus$), such that:

- (A,1,$\otimes$) is a monoid;
- (A,0,$\oplus$) is a symmetric monoid;
- 0 is the zero of $\otimes$;
- $\otimes$ distributes over $\oplus$, that is,

$$[x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)]$$

and

$$[(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)]$$

Our first typeclass implementation in Haskell

Perhaps the most suitable programming language paradigm to represent mathematical structures is the functional one.

```
class Semiring a where
    zero   :: a
    one    :: a
    (<+>)  :: a -> a -> a
    (<.>)  :: a -> a -> a
    srsum  :: [a] -> a
    srsum   = foldr (<+>) zero
    srprod :: [a] -> a
    srprod  = foldr (<.>) one
```

Extending the semiring to *semiring

Because we are going to use regular expressions to annotate our paths, we can rely on Regular Algebra or Kleene Algebra, that is

```
class (Semiring a)=> StarSemiring a where
    star    :: a -> a
    star  a  = one <+> plus a
    plus    :: a -> a
    plus  a  = a <.> star a
```

As you can notice the two operations are interdefinable

And now . . . the matrix's type

Matrices are not only important as a graph-containers, but also as the "mechanism" to make operations with

```
data Edge i = i :-> i deriving (Eq, Ord, Bounded, Ix)

newtype Matrix i e = Matrix (Array (Edge i) e)

matrix   :: (Ix i, Bounded i)=> (Edge i -> e) -> Matrix i e
matrix f  = Matrix . listArray(minBound, maxBound) . map f $ entireRange

entireRange :: (Ix i, Bounded i)=> [i]
entireRange  = range(minBound, maxBound)
```

The *matrix* function builds a square matrix indexed over all elements of *i* from a given function *f*

A first example: calculating the Transitive Closure

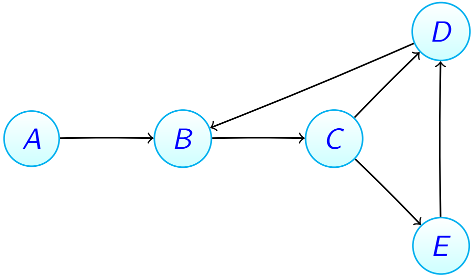
The concept of Transitive Closure can be thought of as constructing a data structure that makes it possible to answer reachability questions. That is, can one get from node s to node t in one or more hops? After the transitive closure is constructed, as depicted in the following figure, we may determine whether node t is reachable from node s. So, in this case node 1 can reach node 4 through one or more hops.

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4
\end{tikzcd}$$

$$\begin{tikzcd}
1 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left=50] & 2 \arrow[r] \arrow[rr, bend left] & 3 \arrow[r] & 4
\end{tikzcd}$$

A first example

Let's use the previous graph as a reference



a directed, unlabelled (and cyclic) graph with 5 nodes and 6 edges

First example implementation

```
data Connection = Connected | Unconnected deriving Eq
type Graph i = Matrix i Connection

graph :: (Ix i, Bounded i) => [Edge i] -> Graph i
graph edgeList = matrix build
 where
  build i | i `elem` edgeList = Connected
          | otherwise         = Unconnected

data Node = A | B | C | D | E deriving (Eq, Ord, Bounded, Ix, Show)

eG :: Graph Node
eG = graph [(A :-> B), (B :-> C), (C :-> D), (C :-> E), (D :-> B), (E :-> D)]
```

Executing our first program, we have

```
GHCi> printMatrix eG

0 * 0 0 0
0 0 * 0 0
0 0 0 * *
0 * 0 0 0
0 0 0 * 0
```

Getting the Transitive Closure from our Semiring

The only thing we need to do here is to apply the *plus* function of our algebraic structure *starsemiring* over the corresponding matrix (the adjacency one). Executing the function, we have

```
        GHCi> printMatrix . plus $ eG
```

```
0 * * * *
0 * * * *
0 * * * *
0 * * * *
0 * * * *
```

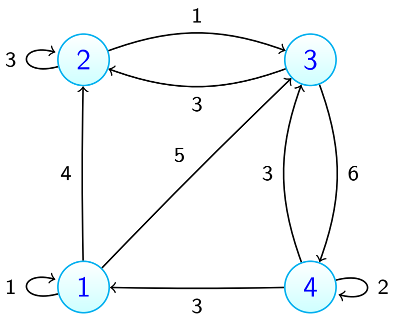
What have we done? We turn the Connection data type into an instance of our Semiring and StarSemiring typeclasses. That is, the additive operation in this instance will be parallel composition of connections and the multiplicative operation will be the sequential composition of connections.

```
instance Semiring Connection where
   zero                  = Unconnected
   one                   = Connected
   Connected   <+> _ = Connected
   Unconnected <+> x = x
   Unconnected <.> _ = Unconnected
   Connected   <.> x = x

instance StarSemiring Connection where
   star _ = one
```

Because for Connections 1 *plus* anything is 1, we do not care about this implementation.

A second example: All shortest-paths



And the adjacency matrix is

$$A = \begin{pmatrix} 1 & 4 & 5 & \infty \\ \infty & 3 & 1 & \infty \\ \infty & 3 & \infty & 6 \\ 3 & \infty & 3 & 2 \end{pmatrix}$$

New requirements for the second example

So far, we can apply operations over unlabelled graphs. In other words, we can only decide whether two are connected or not. Now we need new features to handle *labels* over the edges of the graphs in order to compute *paths*.

There could be an infinite number of paths from a given starting node to a target node.

Fortunately we can represent this infinite number of paths in a finite way, that is, using *regular expressions*, the main reason of the creation of the **semiring*

data type for Expressions

```
data StarSemiringExpression a
    = Var a
      | Or   (StarSemiringExpression a) (StarSemiringExpression a)
      | Seq  (StarSemiringExpression a) (StarSemiringExpression a)
      | Star (StarSemiringExpression a)
      | None
      | Empty
```

Now can create our *type* and our *function* for regular expressions

```
newtype RE a = RE (StarSemiringExpression a)

  re  :: a -> RE a
  re   = RE . Var
```

Instances for regular expressions

```
instance Semiring (RE a) where
  zero                          = RE None
  one                           = RE Empty
  RE None      <+>  y           = y
  x            <+>  RE None     = x
  RE Empty     <+>  RE Empty    = RE Empty
  RE Empty     <+>  RE (Star y) = RE (Star y)
  RE (Star x)  <+>  RE Empty    = RE (Star x)
  RE x <+> RE y                 = RE (x `Or` y)

  RE Empty     <.>  y           = y
  x            <.>  RE Empty    = x
  RE None      <.>  _           = RE None
  _            <.>  RE None     = RE None
  RE x         <.>  RE y        = RE (x `Seq` y)

instance StarSemiring (RE a) where
  star (RE None)       = RE Empty
  star (RE Empty)      = RE Empty
  star (RE (Star x))   = star (RE x)
  star (RE x)          = RE (Star x)
```

Need for more than regular expressions . . .

The above implementation give us the whole regular expression that exists between two nodes, but is huge! For instance, the string representing the position a_{44} of our graph contains 4,867 characters!!.
But we are still looking for the shortest path, we need new constructors

```
data Tropical a = Tropical a | Infinity deriving (Eq, Ord)
```

The corresponding Instances of the semiring for Tropical data type

This semiring has *minimum* as the additive operation, and *addition* as the multiplicative operation, we add $+\infty$ as the identity element for *minimum*

```
instance (Ord a, Num a) => Semiring (Tropical a) where
  zero                            = Infinity
  one                             = Tropical 0
  Infinity     <+> y              = y
  x            <+> Infinity       = x
  (Tropical a) <+> (Tropical b) = Tropical (min a b)

  Infinity     <.> _              = Infinity
  _            <.> Infinity       = Infinity
  (Tropical x) <.> (Tropical y) = Tropical (x + y)
```

In a tropical semiring the multiplicative identity, one, is 0 which is the smallest element possible. This means the asteration operation is the constant one.

```
instance (Ord a, Num a) => StarSemiring (Tropical a) where
  star _ = one
```

Do we have some results? Yes, But . . .

```
GHCi> printMatrix . star . fmap(maybe zero Tropical) $ eG2

                0     4     5     11
                10    0     1     7
                9     3     0     6
                3     6     3     2
```

That is, All shortest paths, but . . ., only their numeric values (minimal distances), *not* the *paths*.

So we need to keep track on both things, numerical values as well as paths

```
data ShortestPath a b = ShortestPath (Tropical a) b
```

The corresponding Instances of the semiring for ShortestPath data type

When we compute the additive operation of the shortest path we will take ancillary data corresponding to the smaller tropical value. In case both tropical values are equal, then we take both pieces of ancillary data together by adding them. The multiplicative operation is simply lifted from the tropical operation and the multiplicative operation on the ancillary data.

```
instance (Ord a, Num a, Semiring b) => Semiring (ShortestPath a b) where
   zero                            = ShortestPath zero zero
   one                             = ShortestPath one one
   ShortestPath a x <+> ShortestPath b y
                    | c < b        = ShortestPath a x
                    | c < a        = ShortestPath b y
                    | otherwise    = ShortestPath c (x <+> y)
                  where  c = a <+> b

   ShortestPath a x <.> ShortestPath b y
                                   = ShortestPath (a <.> b) (x <.> y)
```

...

The star operation simply returns one (which is the tropical value 0) in almost all cases. However, when the tropical value is already one (which is the tropical value 0), we can freely sequence this value as many times as we want. Therefore, in this case we return the asteration of the ancillary data.

```
instance (Ord a, Num a, StarSemiring b) =>
                              StarSemiring (ShortestPath a b) where

    star (ShortestPath x b) | x == one  = ShortestPath one (star b)
                            | otherwise = ShortestPath one one
```

Now, we can 'attach' or annotate our regular expressions as edge names and compute only the shortest paths

```
annotate :: (Ix i, Bounded i, Ord a, Num a, Semiring b) =>
     ((Edge i) -> b) -> Matrix i (Maybe a) -> Matrix i (ShortestPath a b)
annotate f m = go <$> m <*> labelGraph (connect m)
  where
    go v e = ShortestPath (maybe zero Tropical v) (maybe zero f e)
```

Shortest Path with 'minimal distance' and its 'traced-path'

Running our new function

```
GHCi> printMatrix . star . annotate re $ eG2
[0]                  (12)[4]          (13)|(12)(23)[5]    (13)(34)|(12)(23)(34)[11]
(23)(34)(41)[10]     [0]              (23)[1]             (23)(34)[7]
(34)(41)[9]          (32)[3]          [0]                 (34)[6]
(41)[3]              (43)(32)[6]      (43)[3]             [0]
```

A new problem arises, which path we should choose when there is more than one?

Which path?

One approach to include some criteria at a time is the so-called *lexicographical combination*, because it can be applied to numbers or strings, in fact, it is included into *multicriteria* path-finding problems.

So, first we going to translate our last results into the language obtained by the regular expressions

```
newtype Language a = Language [[a]] deriving Show
```

Instances for Language data type

```
instance Semiring (Language a) where
   zero                              = Language []
   one                               = Language (pure [])
  (Language x) <+> (Language y) = Language (x `interleave` y)
    where
       []     `interleave` ys = ys
       (x:xs) `interleave` ys = x:(ys `interleave` xs)

  (Language x) <.> (Language y) = Language (dovetail (++) x y)
    where
       dovetail f l1 l2 = concat $ go l1 (scanl (flip (:)) [] l2)
         where
            go [] _            = []
            go l1 l2@(x:y:ys) = (zipWith f l1 x):(go l1 (y:ys))
            go l1@(a:as) [x]  = (zipWith f l1 x):(go as [x])

instance StarSemiring (Language a) where
   star (Language l) = one <+> plusList (filter (not . null) l)
     where
        plusList [] = zero
        plusList l  = star (Language l) <.> (Language l)
```

Example with the paths as words of the Language

We need now the function *wword* to handle the words of the language generated by the regular expressions and just after that decide which path to stick with

```
wword x = Language [[x]]
```

```
GHCi> printMatrix . star . annotate wword $ eG2
```

```
[[]][0]                    [[(12)]][4]         [[(13)],[(12),(23)]][5]    [[(13),(34)],[(12),(23),(34)]][11]
[[(23),(34),(41)]][10]     [[]][0]             [[(23)]][1]                [[(23),(34)]][7]
[[(34),(41)]][9]           [[(32)]][3]         [[]][0]                    [[(34)]][6]
[[(41)]][3]                [[(43),(32)]][6]    [[(43)]][3]                [[]][0]
```

Last example

Finally, the function that extract words from the language and gets the *smallest* according to the *lexicographical combination*

```
    minimalWord (Language l)
                | length l > 1 = [minimum l]
                | otherwise    = l
```

```
GHCi> printMatrix . fmap(minimalWord . extract) . star . annotate wword $ eG2
```

```
[[]][0]                  [[(12)]][4]        [[(12),(23)]][5]   [[(12),(23),(34)]][11]
[[(23),(34),(41)]][10]   [[]][0]            [[(23)]][1]        [[(23),(34)]][7]
[[(34),(41)]][9]         [[(32)]][3]        [[]][0]            [[(34)]][6]
[[(41)]][3]              [[(43),(32)]][6]   [[(43)]][3]        [[]][0]
```

Further Work

As we have seen in the last two examples, Multicriteria path-finding problems face a variety of challenges when the criterion comes from different path algebra, for instance, the choice between a Shortest path and a Enumeration of elementary paths: the Set for Shortest path is $\mathbb{R}^+ \cup \{+\infty\}$ while the Set for the Enumeration path is $\{0, 1\}$.

Finally, the development of language features to support these algebras as parameters.

References

Günter Rote, Path Problems in Graphs

Robert Endre Tarjan, A Unified Approach to Path Problems,

Roland Backhouse and A.J.M. van Gasteren, Calcultating a Path Algorithm

Roland Backhouse and B.A. Carré, Regular Algebra Applied to Path-finding Problems,

Michel Gondran and Michel Minoux, Graphs, Dioids and Semirings,

Roland Bachouse, Algorithmic Problem Solving

Russell O'Connor, Shortest Path in Haskell, http://r6.ca/