# CS 613 Functional Programming

Course Project

# Graphs in functional Programming

Excerpt from the paper

First, we want to demonstrate that it is possible to dene graph algorithms in a distinctive functional style and that these algorithms are at the same time often competitive in terms of eciency with typical imperative implementations. Second, by giving a collection of algorithms typically found in courses on algorithms and data structures, we try to close a gap in functional algorithms and data structure textbooks.

# Graph Constructors

The Different types of Data

Context is the (edges falling on it, Number of node, “label of node”, edges originating from the node)

$$
\begin{aligned}
&\textbf{type } Node &&= Int \\
&\textbf{type } Adj\ b &&= [(b, Node)] \\
&\textbf{type } Context\ a\ b &&= (Adj\ b, Node, a, Adj\ b)
\end{aligned}
$$

# Graph Data

The Graph Data type as is given in the paper

$$\mathbf{data}\ Graph\ a\ b\ =\ Empty \mid Context\ a\ b\ \&\ Graph\ a\ b$$

# My implementation of the Graph Class

I need to define the infix operator “:&”

The Graph need to be explicitly defined as right Associative

```
data Graph a b = Empty | (Context a b) :& Graph a b
               deriving (Eq, Show)
```

# Utility Functions for The Graph Algorithms
# My Implementations

# isEmpty

The isEmpty Function returns true if the Graph is same as the empty constructor
And it returns false if not

```haskell
isEmpty :: Graph a b -> Bool
isEmpty Empty = True
isEmpty _ = False
```

# gmap

The Map function for graphs that is used to apply a function to all the nodes of the graph

The pattern matching used as c :& g

```
gmap :: (Context a1 b1 -> Context a2 b2) -> Graph a1 b1 -> Graph a2 b2
gmap f Empty = Empty
gmap f (c :& g) = f c :& gmap f g
```

# grev

Reverse the edges in the graph changes the predecessor and the successor of the Graph

```haskell
grev :: Graph a b -> Graph a b
grev = gmap swap
    where swap (p, v, l, s) = (s, v, l, p)
```

# Ufold (unordered Fold )

A fold function for the Graph where the order of the nodes is not important

```
ufold :: (Context a b -> t -> t) -> t -> Graph a b -> t
ufold f id Empty = id
ufold f id (c :& g ) = f c (ufold f id g )
```

# nodes

It returns the nodes that are present in the given graph

```haskell
nodes :: Graph a b -> [Node]
nodes = ufold (\(p, v , l , s ) -> (v:)) [ ]
```

# undir

Adds a edge in  each direction for each directed edge to essentially create a undir edge in the graph

```
undir :: (Eq b) => Graph a b -> Graph a b
undir = gmap (\(p,v, l, s) -> let ps = nub (p++s) in (ps, v, l, ps ))
```

# Match

Match is one of the most important function in the library
It takes as input a graph and a node and returns the context of the node that is all the edges and the label and also the remaining graph after removing the node

I have implemented a simpler version of the match function that is does not change the other contexts and just finds out the context corresponding to the node

This prevents the graph algorithms to work on starting from all nodes and we need to start from the first node in the structure

Though no error occurred in testing it might be prone to errors

# Match v2.0

The Better Match Function

Creating this match function which is entirely different from the one given in the FGL Library might be a start to the new library

This match is based on the use of defining Graphs in another way
As a pair of lists of labelled Nodes and Labelled Edges

```
type Graph2 a b =  ([LNode a], [LEdge b])
```

The Graphs2 are supposed to be an internal feature of the code and not supposed to be used by the user

They are based on how Graphs are denoted in Discrete Mathematics as

G = (V, E)

V => Set of Vertices

E => Set of Edges

# Graph12

A Function to convert Graph of first Type to Graph of the second Type

Is used as a utility Functions later for various other Functions

```haskell
lnodes = ufold (\(p, v , l , s ) -> ((v,l):)) [ ]

ledges g = concat (ufold (joi) [] g)
    where
        joi (p,v,l,s) = (((foldr (pre v) [] p ) ++ (foldr (nex v) [] s)) :)
        pre v (l', v') = ((v, v',l'):)
        nex v (l' ,v') = ((v',v,l') :)

graph12 :: Graph a b -> Graph2 a b
graph12 g = (lnodes g, ledges g)
```

## lnodes

Returns a list of labelled nodes in the graph

## ledges

Returns a list of labelled Edges in the graph

# Graph21

A function to change the nodes and edges graph back to the list of context graph

```
graph21 g@(lnodes, ledges) = trim (foldr (expand ledges) Empty lnodes )
```

# Expand

The Inner part of the graph21 function essentially creates a graph with all context filled up that is all context corresponding to all nodes have all the edges falling on it and all the edges emerging from it

```
expand :: [LEdge a] -> LNode b -> Graph b a -> Graph b a
expand ledges lnode@(a, b) =  ((toAdj1 lnode ledges , a , b , toAdj2 lnode ledges) :&)
```

# Trim

The Trim function reduces the expanded list of contexts to the correct version where once a node is selected it has all the edges remaining in its context and then these edges are not used in other contexts

```haskell
trim Empty = Empty
trim (c@(p, v, l, s) :& g) = c :& (myrem v (trim g))
myrem v g = gmap (rem' v) g

rem' v' (p, v, l, s) = (p', v, l, s')
 where
    p' = filter (noOccur v') p
    s' = filter (noOccur v') s

noOccur v (l, n)
  | v == n = False
  | otherwise = True
```

# Now the New Match Function

```haskell
match :: Node -> Graph a b -> (Maybe (Context a b), Graph a b)

match n Empty = (Nothing, Empty)
match n g
    | isThere n g = (Just (graphContext n (graph12 g)), remNode n g)
    | otherwise = (Nothing, Empty)
```

# Isthere

Isthere Function returns true if the required node is in the graph else it returns false

```
isThere n Empty = False
isThere n ((p, v, l, s) :& g)
    | v == n = True
    | otherwise = isThere n g
```

# GraphContext

This Function takes out the full Context of the given node (that is all the edges which have this node as a vertex) from the given Graph

```
graphContext n g@(lnodes, ledges) = myFirst (myfilter ( \(p,v,l,s) -> (v==n) )
                                            (foldr (expand ledges) Empty lnodes))
```

# remNode

This function returns the remaining graph after the required node is removed from it

```
remNode v g =  gmap (rem' v) (myfilter (\(p,v',l,s)-> (v /= v')) g)
```

# dfs

The DFS work on the principle that in each iteration of the function, we add the successor of the node to the front of the queue, and remove the node from the graph
Then in next iteration we will use the successor of the previous node

```
dfs [] g = []
dfs (v:vs) (g) = case match v g of
                 (Just c, g) -> v : dfs (suc c ++ vs ) g
                 (Nothing, g) -> dfs vs g
```

# bfs

The BFS similar to the DFS work on the principle that in each iteration of the function, we add the successor of the node to the last of the queue, and remove the node from the graph
Then in the following iterations we will use all the successors before going to the next set of distance

```
bfs [] g = []
bfs (v:vs) (g) = case match v g of
                 (Just c, g) -> v : bfs (vs ++ suc c ) g
                 (Nothing, g) -> bfs vs g
```

# Shortest Path

The Bft function returns all the paths from the node v to all the other paths
In case of a loop the function returns an infinite number of paths

```
bft v = bfsPath [[v]]

bfsPath :: [Path] -> Graph a b -> RTree
bfsPath [] g = []
bfsPath (p@(v:_):ps) g = case match v g of
                           (Just c , g') -> p: bfsPath (ps ++ map (:p) (suc c)) g
                           (Nothing, g') -> bfsPath ps g
```

The esp function first says haskell to calculate the bft from the source node s

Then it asks for the smallest path to t

It is possible due to laziness of Haskell that when the first value is found the program stops working

```
esp :: Node -> Node -> Graph a b -> [Node]
esp s t = reverse . first (\(v:_)-> v== t). bft s
```

# Independent Set

```haskell
indep Empty = []
indep g = if  length i1 > length i2 then i1 else i2
          where
              vs = nodes g
              m = maximum (map (flip deg g) vs )
              v = first (\v -> deg v g == m) vs
              (Just c, g') = match v g
              i1 = indep g'
              i2 = v: indep(foldr del g'(pre c ++ suc c))
```

The Independent Set is calculated as we would calculate it in a dynamic programming problem
Create two different sets one on selecting a node one with including the node and one without including that node

Solve the problem Dynamically/ Recursively

Check which set has the higher amount of nodes and return it as the answer

# References

- https://www.researchgate.net/publication/2364482
  Inductive Graphs and Functional Graph Algorithms
- https://github.com/haskell/fgl/blob/master/Data/Graph/Inductive/Graph.hs
  GitHub Repository of the FGL library
- Hackage
- Hoogle

Thank You

– Sanchit Jindal