Inductive Graphs and Functional Graph Algorithms

Author : Martin Erwig
Reviewed by : Himanshu Singh (16305R005), Vikas Kumar(163059009)

It's quite a different thing to somwhow write a graph algorithm and an efficient graph algorithm. Difficulty arises when a node is reached by more than one node and algorithm requires it to be visited at most one time. In imperative language, this task is accomplished by mainting an extra visited array. In functional programming, similar thing can be achieved by passing a list to a function which maintains the state for each node if it isvisited or not. But this method suffers with two problems : 1.) Efficiency : If we use balanced binary search tree for storing the state of each node, insertion and membership query will take O(logn) time unlike imperative language where it takes constant time. 2.) Clarity : If this data structure is shared by many functions then it has to be passed in each function as an argument which spoils the readability of the program. This is worse because it unnecessarily makes it difficult to prove the property of the program.

Work done in this paper

Inductive Graph

Lists and trees algorithms have simple definitions and do not need any kind bookkeeping. This is because lists and trees are defined as inductive data structures. This paper focuses on representing a graph as an inductive data structure. Graph is represented by two constructors. Some simple algorithms can be represented by these two constructors using pattern matching. More advanced algorithms require nodes to be visited in specific order which is supported by a particular kind of pattern matching which will be described later in this review.

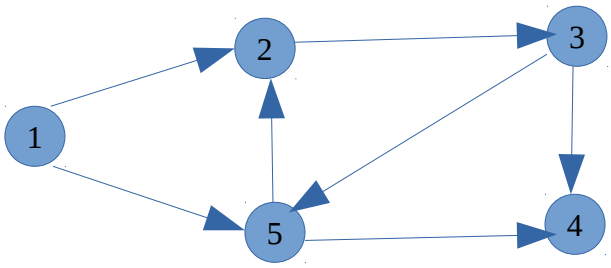
Graph Constructor

In this paper, graph vertices are represented by integers for simplicity. For generality, a single graph type is i.e. graph type for directed multi graphs where nodes and edges are labelled. All other graph types e.g. undirected graph, unlabelled graphs etc can be defined by putting some restrictions on the above generalized definition. The inductive definition of graph is captured by following cases: a graph is an empty graph or a graph extended by a vertex v with it's predecessors and successors that are already present in the graph. This one-step inductive graph extension is defined by a type called **Context** :

```
type Node          = Int
type Adj b         = [(b, Node)]
type Context a b   = (Adj b, Node, a, Adj b)
```

and graph is defined using **Context** and another constructor **Empty**

```
data Graph a b = Empty | Context a b & Graph a b        (& : infix notation)
```



Let’s assume edge from vertex u to v be named as “u->v” and Label for vertex i be vi. This graph can be build as
([], 1, ‘v1’, [(“v1->v2”, 2), (“v1->v5”, 5)]) &
([(“v5->v2”, 5)], 2, ‘v2’, [(“v1->v2”, 2), (“v1->v5”, 5)]) &

An error is reported when a context for a node is added and it is already present in the graph. An error is also reported when a node mentioned in the successor or predecessor list is missing in the graph. While creating the graph we can chose node in any order for insertion. This gives a powerful kind of pattern matching on graphs which will be described later. This result can be expressed by following to facts:
1.) **Completeness** : Each labeled multi-graph can be represented by a graph term.
2.) **Choice of Representation** : For each graph g and each node v contained in g there exists some p, l, s and g’ such that (p, v, l, s) & g’ denotes g.
Some useful operations are defined to make the task of forming a graph by contexts easier.
NewNodes : This function is useful for extending the graph which construction history is not known.
newNodes :: Int -> Graph a b -> [Node]
newNodes i g = [n+1..n+i] where n = foldr max 0 (nodes g)
nodes : This function extracts the node values from a graph.
e.g. let 5 be the largest value assigned to a vertex in the graph then if we want to add 3 more nodes to the graph then it’s value should be [5+1..5+3] i.e. 6,7,8.

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

map function for graph :

```
gmap :: (Context a b -> Context c d) -> Graph a b -> Graph c d
gmap f Empty = Empty
gmap f (c & g) = f c & gmap f g
```

gmap preserves the strucure of the node but not necessarily structure of edges. For example, reverse graph can be constructed using gmap hence structure of edges is changed.

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

Here in *swap* function, predecessor and successor are swapped for each node.
This style of programming makes proofs of program properties easier. For example, we can prove *fusion law for gmap* and an *inversion rule for grev* in couple of lines.

gmap f . gmap f' = gmap (f . f') *(gmap fusion)*
grev . grev = id *(grev inversion)*

gmap fusion proof :

```
gmap f ( gmap f' g ) = gmap f ( gmap f' ( c & g' ) )          (Def. of g)
                     = gmap f ( f' c & ( gmap f' g' ))        (Def. of gmap)
                     = f (f' c) & gmap f (gmap f' g')         (Def. of gmap)
                     = (f . f') c & gmap (f . f') g'          ( Inductive Hypotheses )
                     = gmap (f . f') (c & g')                 (Def. of gmap)
                     = gmap (f  . f') g                       (Def. of g)
```

grev proof :

To prove *grev inversion law,* we need two obvious facts about *swap* and *gmap* function :

swap . swap = id *(swap has idempotency property)*
gmap id = id *(gmap unit)*

To prove grev inversion, we'l use gmap fusion law.

```
grev . grev = gmap swap . gmap swap      (Definition of grev)
            = gmap (swap . swap)         (gmap fusion law)
            = gmap id                    (swap idempotency)
            = id                         (swap unit)
```

ufold (unordered fold which means order of encountering nodes is not important) function for graph:

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

ufold can be used to define gmap function as follows:

```
gmap f = ufold (\c -> (f c & )) Empty
```

List out all nodes of a graph.

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

Make the graph undirected.

```
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))
```

Since graphs are implemented in Functional Graph Library (FGL) as an abstract type, & is a function and not a constructor and therefore can not be used in patterns.

Active Pattern : It extends patterns by a function component that is applied to the argument value before it is matched against the pattern. What we need is the matching and extracting of pattern matching, but with the added ability to transform the data we're working with. Active Patterns are special kind of function with some extra powers that make them useful in pattern matching.

Active Graph Pattern : Choice of Representation tells that for each node v contained in a graph there is a term representation (p, v, l, s) & g for some suitable p,l,s and g. Now active pattern (c $\&^v$ g) is matched against a graph g' by searching for node v in g' and transforming g' into a term representation in which v's context is inserted last, so that it is the argument of the outermost application of &. This can be done only if v is contained g'. In that case, pattern is said to match and v's context is bound to c and the graph without the context i.e. without v and its incident edges is bound to g. On the other hand, if v is not contained in g', the patterns fails, no bindings are produced and pattern matching continues as after a normal pattern-matching failure.

Active graph patterns are used to determine : node's successor, computing the degree of a node or deleting a graph from node.

Successor of a node in the given graph:

```
gsuc :: Node -> Graph a b -> [Node]
gsuc v ((_, _, _, s) &^v g) = map snd s
```

Degree of a node:

```
deg :: Node -> Graph a b -> Int
deg v ((p, _, _, _) &^v g) = length p + length s
```

Delete a node from the graph

```
del :: Node -> Graph a b -> Graph a b
del v (_ &ᵛ g) = g
```

Implementation and Complexity

Implementation of inductive graphs should support following operations for constructing and decomposing graphs.

Table 1. *Basic Graph Operations*

Construction		Decomposition	
Empty graph	(*Empty*)	Test for empty graph	(*Empty*-match)
Add context	(&)	Extract arbitrary context	(&-match)
		Extract specific context	($\&^v$)

Graphs have to be fully persistent i.e. updates on graph must leave previous versions intact.

Graph Representation and Persistence

In imperative language, a graph is represented by adjacency list or adjacency matrix. Adjacency list is preferred over adjacency matrix on most occasions. So, in this paper discusses two alternatives for making adjacency list persistent.

Version Tree Representation : Quite Complex to understand.

Binary Search Tree Representation : A graph is represented by a pair (t,m) where t is a tree of pairs (node, (predecessors, label, successors)) and m is the largest node occurring in t. This m is used to support the creation of new nodes in the graph in constant time.

However, inserting and deleting a node context (p, v, l, s) takes much effort. For insertion, we have to insert the context itself which takes O(logn) steps and we have to insert v as a successor (predecessor) for each node in p (s) which requires O(clogn) steps which is as large as O(nlogn) steps for dense graphs. Context deletion takes even more time since we have to remove v from successor (predecessor) list for each element of p (s) which requires searching these lists for v. Hence, deletion runs in $O(c^2 logn)$ time or O(clogclogn) time if predecessors and successors are stored as search trees. In dense graph, this gives time complexity of $O(n^2 logn)$ or $O(nlog^2 n)$. Although asymptotic time complexity of binary search tree representation is worse than array representation but in practice it performs better might be beacuse of simplicity and does not require much tuning.

Functional Graph Algorithms

Depth First Search

In depth first walk, each node is visited once by visiting successors before siblings. The parameters of depth-first search are the graph to be searched and a list of nodes saying which nodes are left to search. This list is needed for unconnected graph

where aftering exploring one component, a node of another component is needed to continue the search.

```
dfs :: [Node] -> Graph a b -> [Node]
dfs []      g            = []
dfs (v:vs)  (c &v g)     = v:dfs (succ c ++ vs) g
dfs (v:vs)  g            = dfs vs g
```

Working of the algorithm : If there are no nodes left to be visited (first case), dfs stops without returning any nodes. In contrast, if there are still nodes that must be visited, dfs tries to locate the context of the first of these nodes (v) in the argument graph. If this is possible (second equation), which is the case whenever v is contained in the argument graph, v is the next node on the resulting node list, and the search continues on the remaining graph g with the successors of v to be visited before the remaining list of nodes vs. The fact that the successors are put in front of all other nodes causes dfs to favour searching in the depth and not in the breadth. Finally, if v can not be mathced (last line), dfs continues the search with the remaining list of nodes vs. Last will occur only if v is not contained in the graph because otherwise the pattern in second equation would have matched.

There is a source of optimization in the above definition: One can immediately terminate dfs and return an empty node list when nodes to be visited are still left but the graph is empty. This can be done by either adding following as the first or second equation :

```
dfs vs Empty = []
```

or changing the first equation to following :

```
dfs vs g | null vs | isEmpty g = []
```

Breadth First Search

BFS essentially means visiting sibling before successors.

```
bfs :: [Node] -> Graph a b -> [Node]
bfs []        g          =  []
bfs (v:vs) (c &v g)      =  v:bfs (vs ++ suc c) g
bfs (v:vs) g             =  bfs vs g
```

This algorithm works in the same way as that of dfs except for the treatment for the newly found successors and this reflects exactly the way in which the nodes to be visited are implicitly ordered. For dfs, these nodes are kept in stack and for bfs these nodes are kept in queue.

Algorithm for Splitwise's Simplify Feature :

Step 1 : Find out net amount of money one has to pay i.e. it is positive for a user if he has to pay to someone and negative if he is getting from someone.
Step 2 : Make two lists : 1.) Giver : consists of person who has to pay someone when net balance is calculated 2.) Taker : consists of person who has to take money from someone when net balance is calculated.
Step 3 : Sort both lists.
Step 4 : Make a bipartite graph where one set of users belong to Giver and other set belongs to Taker.
Step 5 : First make an edge between two users if they exactly satisfy the demand and supply. Take first user from Giver and Taker, if Giver has to give more than the Taker has to take then make an edge from Giver to Taker with label as the amount that Taker has to take. Increment the pointer in Taker list. Keep pointer in Giver list at the same position while updating the new amount for Giver as (Giver – Taker). Similarly, if Giver has to give less than the Taker has to take then make an edge from Giver to Taker with label as the amount that Giver has to give. Increment the pointer in Giver's list while keeping the pointer in taker list at the same position updating it's new amount to (Taker-Giver). If amount that Giver has to give and Taker has to take are exactly same then make an edge from Giver to Taker with edge label as the amount and increment the pointer in both lists.
Step 6 : Continue like Step 5 till both lists are empty.

This algorithm guarantees that there will be atmost (|V|-1) number of transactions because the graph is acyclic.