A Functional Graph Library

Christian Doczkal

Universität des Saarlandes

Abstract. Algorithms on graphs are of great importance, both in teaching and in the implementation of specific problems. Martin Erwig proposes an inductive view on graphs to achieve a concise notation, persistence and efficiency. I will show that, though meeting each of these goals, the proposed solution fails to archive all three at the same time, especially when extending the term “efficiency” beyond time complexity.

1 Introduction

I will provide an overview and and an evaluation of the inductive view on graphs and its implementation as presented in [Erw01]. Section 2 covers the inductive graph model and presents active patterns which are very convenient when dealing with inductive graphs. In section 3, two possible implementation ideas for the inductive graph model are briefly discussed and compared. In section 4, I will give an example of an inductively defined graph algorithm and comment on its expressiveness, efficiency, and suitability for programming and also for teaching, which is also one of Erwig's goals.

2 Inductive Graph definition

Functions defined over inductive data types like trees are usually very expressive and elegant. By providing an inductive definition for graph data structures, those benefits can also be applied to functional graph algorithms. Erwig proposes such an inductive definition a well as an implementation that allows inductive graph algorithms to be implemented with the same asymptotic complexity as their imperative counterparts. The paper only deals with directed, node and edge labeled multi graphs, since other graph types are merely special cases of this definition. Intuitively, inductive graphs can be seen as an algebraic data type defined as follows, where & is an infix constructor (grouping right).

$$\begin{array}{ll}
\mathit{type\ Node} & = \mathit{Int} \\
\mathit{type\ Adj\ b} & = [(b, \mathit{Node})] \\
\mathit{type\ Context\ a\ b} & = (\mathit{Adj\ b},\ \mathit{Node},\ a,\ \mathit{Adj\ b}) \\
\mathit{data\ Graph\ a\ b} & = \mathit{Empty} \mid \mathit{Context\ a\ b}\ \&\ \mathit{Graph\ a\ b}
\end{array}$$

The above definition uses the invariant that upon insertion into the graph, every node that is mentioned in the context of the newly inserted node must already be present in the graph. Although this term representation is insufficient for implementation purposes, due to the invariant and the need to efficiently retrieve specific nodes, it is a good intuition for writing algorithms. Figure 1 shows a graph and one possible term representation. First the solid then the dashed and finally the dotted elements are inserted.

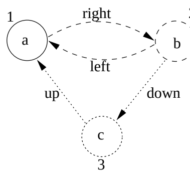

Fig. 1. Inductive Graph with term representation

$$
\begin{array}{ll}
([(\text{``}\mathit{down}\text{``},\ 2)],\ 3,\ 'c',\ [\text{``}\mathit{up}\text{``},\ 1)])\ \& & \\
([(\text{``}\mathit{right}\ \text{``},\ 1)],\ 2,\ 'b',\ [\text{``}\mathit{left}\text{``},\ 1)])\& & \\
([],\ 1,\ 'a',\ []) & \&\ \mathit{Empty}
\end{array}
$$

The above definition also provides two important facts that are important for implementation purposes and for the definition of a new kind of pattern matching called *active patterns*, which provides a very concise notation for graph algorithms.
Fact 1(Completeness) : Each labeled multi-graph can be represented by a graph term.
Fact 2 (Choice of Representation):For each graph g and each node v contained in g there exist p, l, s, and g' such that $(p, v, l, s)\&g'$ denotes g.

2.1 Active Patterns

Fact 2 states that a long as v is present in g, the graph can be separated into vs context and the graph that remains if v and all edges adjacent to v are removed from the graph. We now define the graph primitive $\&^v$-*match* , which conceptually performs this transformation and returns $(\mathit{Just}\ c,\ g')$ or $(\mathit{Nothing},\ g)$ if v was not present in the graph. Using $\&^v$-*match* , we define an extension to Haskell's pattern matching feature called active patterns. The pattern $(c\ \&^v\ g)$ is matched if $\&^v$-*match* $v\ g$ returns $\mathit{Just}(c, g)$. This allows a very concise notation for graph algorithms. Although being neat, active patterns are merely a syntactic extension of current Haskell. They can also be expressed using a case construct or *pattern guards* as provided by the current ghc implementation. The function f, which uses our active pattern

$$
\begin{array}{ll}
f\ p\ (c\ \&^v\ g) & = e \\
f\ p\ g & = e'
\end{array}
$$

where v is contained in the pattern p could be reformulated
... using the case construct ... or alternatively using pattern guards

$$
\begin{array}{lll}
f\ p\ g' & = & \textbf{case}\ \mathit{match}\ v\ g'\ \textbf{of} \\
 & & (\mathit{Just}\ c,\ g)\ \rightarrow\ e \\
 & & (\mathit{Nothing},\ g)\ \rightarrow\ e'
\end{array}
\qquad
\begin{array}{ll}
f\ p\ g'|(\mathit{Just}\ c, g)\ \leftarrow\ \mathit{match}\ v\ g' & = e \\
\qquad |g\ \leftarrow\ g' & = e'
\end{array}
$$

3 Implementation

The implementation of inductive graphs as an algebraic data type forbids itself, since graph decomposition ($\&^v$-*match*) would require the graph to be transformed. Furthermore the & constructor could be used incorrectly violating the graph invariant. Since the graph data structure is designed for the functional world, the graph implementations should be fully persistent. Additionally we want the graph implementation to allow graph algorithms to be implemented with the same asymptotic complexity as their imperative versions. Erwig provides two different implementations. One uses binary search trees, the second implementation uses a rather complex data structure based on array version trees. I will give a brief overview over both of them. A more detailed description can be found in the original paper. Both give reasonably efficient implementations for the graph primitives from the following table:

Construction	**Decomposition**
& (add context)	&-*match* (extract arbitrary content)
Empty (empty graph)	$\&^v$-*match* (extract specific content)
	Empty-match (test for empty graph)

3.1 Binary Search Tree Representation

In this model, Graphs are implemented as a balanced binary search tree containing all contexts present in the graph in the from
$(node, (predecessors, label, successors))$ The balancing and search tree invariant are based on the node integer. For efficiency reasons, the complete predecessor and successor lists are stored in each node. This leads to a double representation of edges, but allows contexts to be matched without changes and to directly report all contexts that need to be modified when a node is deleted from the graph.

As an optimization of this structure, the predecessor and successor lists are also stored as binary search trees. Furthermore we pair our tree with a single integer containing the highest node present in the graph which is used to report new nodes. Using this representation, *Empty* and *Empty-match* are easily implemented. $c\&g$ works by first inserting the context into the tree and then inserting every successor/predecessor into the corresponding nodes predecessor/successor list. $\&^v$-*match* works the other way around, locating the context to be matched, deleting v from the adjacency lists of all related nodes and returning the context alongside the remaining graph.

Considering the time complexity of the operations above we define n as the number of vertices in the graph, m as the number of edges and c_v as the size of vs context ($|pred\ v| + |suc\ v|$), where *pred* and *suc* denote predecessor and successor functions. Finally c denotes the biggest context relevant to one operation. Using these definitions node insertion and deletion and the *match* functions based on them run in $O(c_v \log n \log c)$

3.2 Array Version Tree Representation

This implementation builds upon the idea of array version trees, an implementation for functional arrays. Instead of a in tree the contexts are stored in an array indexed by the node integer. Any changes to the array are recorded using an inward directed tree of $(index, value)$ pairs, which has the original array as its root. This way the different array versions can be represented as pointers to nodes in the tree, and the nodes along the path to the root mask older versions of the array. New versions can be added in constant time whereas lookup follows the tree structure up to the original array and may thus take $O(u)$ time, where u is the number of previous updates to the array.

Since the goal is to provide a persistent graph implementation that allows all basic operations except & in $O(1)$ time, the structure needs some optimizations. To allow for any operation ob be in $O(1)$, we need to ensure $O(1)$ read access to the graph array, which we do only for single threaded graph usage. To achieve this, we allocate an imperative cache array which contains a copy of the original array whenever we derive a new version from the version tree root and associate it with the new version. This cache array can be used for $O(1)$ access to the latest version of the graph. When the next version is derived from this version, the cache array is updated as well and handed on. On single threaded graph usage, the version tree degenerates to a linear tree where the latest version is always cached.

As with the tree representation, $\&^v$-*match* and &-*match* are costly operations because they require the deletion of v from the context of every adjacent node. To avoid these costs, we equip every node with a positive integer. This node stamp is also put into the adjacency lists of all adjacent nodes. When calculating adjacency of a node only the nodes where the stamp on the edge matches the stamp on the adjacent node are reported, so all that needs to be done when deleting a node is to negate its node stamp. Upon insertion the node stamp is restored and incremented and the new stamp is taken over to adjacent nodes, so any leftover edges are still not reported. To allow for *Empty-match* and &-*match* to be implemented in $O(1)$ we also store the number of nodes currently present in the graph and keep an additional array (cached in the same manner) which carries a partition of nodes currently in the graph and nodes that are deleted. The node partition can also be used to efficiently report free nodes that can be inserted into the graph.

3.3 Comparison

We now have two implementations for functional graphs. The first is easy to implement, fully persistent, fully dynamic, and reasonably efficient for small and medium size graphs, but clearly does not allow algorithms to be implemented so that they meet imperative time complexity. The second implementation allows graph decomposition and testing in constant time and thus allows graph algorithms to be implemented with the same asymptotic complexity as their imperative counterparts. Unfortunately, it is difficult to implement and the allocation of different cache arrays causes a lot of memory overhead which makes the implementation unsuitable for use in real world applications. Furthermore the structure may be “tricked” by first deriving 2 versions linearly and then continuing with the first version, adding the factor u to the time comlexity for non cached lookup. So already almost single threaded graph usage crashes time bounds.

4 Algorithms and Evaluation

Erwig proposes his graph library for both teaching of graph algorithms and for writing efficient applications. For the purpose of writing practical applications, the library can be of limited use. The binary search tree implementation can be useful since the representation is fully persistent, thus allowing a pure functional stile for working with graphs, while still being fast as log as graphs do not grow to large. This might also be one of the possible reasons to use inductive graphs for teaching graph algorithms. Using the inductive graph implementation described above and the definition for active patterns, depth first search, one of the most important graph algorithms, can be written as follows:

```
dfs                             :: [Node] → Graph a b → [Node]
dfs vs g | null vs || isEmpty g = []
dfs (v : vs) (c &^v g)          = v : dfs (suc c ++ vs) g
dfs (v : vs) g                  = dfs vs g
```

The algorithm works very similar to its imperative counterpart, the biggest difference being the way the single visit constraint is enforced. Imperative implementations tend to use some node marking strategy, either within the graph or in a separate data structure. Since we use an inductive model the single visit constraint is enforced implicitly by decomposing the graph step by step. The algorithm either stops when there are no more nodes to expand or if the whole graph has been traversed and the remaining graph is thus empty. The latter is useful since in dense graphs expanded edges can cause up to $\Omega(n^2)$ nodes to be checked, even after the graph has been fully traversed. The pattern $(c \;\&^v\; g)$ is always matched when v is contained in g. In that case v is expanded. Otherwise, if v has already been expanded, the node is simply discarded. Notably this algorithm can be instantly transformed into a breadth first search algorithm merely by exchanging *suc c* and *vs* in the second equation, although a queue implementation is needed to keep the linear time bound. The original paper shows quite a few standard graph algorithms, which can all be elegantly expressed.

So for the simple teaching of graph algorithms the *fgl* might be useful, as long as time complexity is not a concern. If one really wants in-depth teaching of graph algorithms, one hits the limitations of the library. The current implementation of the *fgl* library for Haskell provides only the binary search tree implementation as described above and monadic graphs which are based upon *IOArrays*. They efficiently allow single threaded graph usage. So one has to chose to either implement algorithms without meeting imperative time bounds or one is left to work within a monad, which completely destroys the functional flavor. Additionally, the documentation of the library has not been updated for the last 4 releases. There is an implementation for SML/NJ which includes both implementations, but it has not been changed since August 1999, and so far I have not been successful in building the library with the current SML/NJ release.

On the other hand, in an impure functional language such as ML, functional graph algorithms can be implemented easily and efficiently using imperative style book keeping. The extra book keeping allows most graph algorithms to be implemented to only read from the graph so a static/functional array representation is entirely sufficient. Furthermore algorithms written in a "read only" style can be implemented in a way so that they appear purely functional to the outside. This approach has been taken in [KL93]. Here, the monad of state transformers is used to efficiently implement the depth first forest algorithm in Haskell. Although notationally considerably less concise, the state of the log is threaded through the different function calls roughly in the same manner as one threads the remaining graph though the recursive function calls when writing inductive style algorithms. The result of this approach is an efficient *dff* algorithm for Haskell that appears completely functional to the outside.

So altogether, the idea to view graphs as inductive data types is a nice idea that theoretically allows for a very concise and elegant notation of functional graph algorithms that do have the same time complexity as their imperative counterparts. Unfortunately one either has to cope with a considerable memory overhead, with unmet imperative time bounds or with a loss of expressiveness and persistence if one decides to use monadic graphs thus destroying the functional flavor. For now, there seems to be no way to achieve persistence, efficiency, and concise and alegant notation all at the same time.

Bibliography

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[KL93] David J. King and John Launchbury. Lazy Depth-First Search and Linear Graph Algorithms in Haskell. In John T. O' Donnell and Kevin Hammond, editors, *GLA*, pages 145–155, Ayr, Scotland, 93. Springer-Verlag.