FGL/Haskell – A Functional Graph Library

User Guide

Martin Erwig
FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

April 12, 2000

Contents

Contents **1**

1 Introduction **2**

2 Inductive Graphs **3**
2.1 Graph Types 3
2.2 Constructing Graphs 4
2.3 Extracting Graph Information 6
2.4 Graph Decomposition 7

3 Basic Graph Operations and Their Implementation **9**
3.1 General-Purpose Operations 10
3.2 Implementation of Graph Operations 11

4 Depth First Search **12**

5 Breadth-First Search **15**

6 Shortest Paths **17**

7 Graph Voronoi Diagram **17**

8 Minimum Spanning Tree **18**

9 Maximum Independent Node Sets **19**

References **20**

Appendix A: Implementation Aspects **22**

Appendix B: Further Reading **25**

Index **27**

1 Introduction

The *Functional Graph Library*, *FGL*, is a collection of type and function definitions to address graph problems. FGL presents an approach to the formulation of graph algorithms that differs significantly from the traditional, imperative way of thinking about and solving graph problems: the basis of the library is an inductive definition of graphs in the style of algebraic data types that encourages inductive, recursive definitions of graph algorithms instead of the traditional node marking style. In addition, the library contains functions for well-known graph problems, such as, depth-first search or computing shortest paths. At present, the library is far from being complete, but we intend to expand it.

FGL is devoted in the first place to enable a concise and clear formulation of graph problems. This means that you should not always expect the most efficient implementations of graph algorithms. Nevertheless, the complexity of functions is an important issue, and algorithms based on the inductive graph view can be, in principle, as efficient as corresponding imperative algorithms. Users most concerned about efficiency should consult Appendix A.

This User Guide aims at providing a novice user quickly with the most important informations to get an impression of FGL and learn how to use the library. We assume familiarity with Haskell and knowledge of basic graph terminology. We will occasionally use functions defined in the 1998 Standard of Haskell [20], otherwise this User Guide should be self-contained. The document is divided into two main parts. The first two sections provide basic information about inductive graphs and how to write recursive graph algorithms:

- In Section 2 we explain the inductive definition of graphs that underlies all algorithms in FGL, and we describe elementary functions to build and to inspect graphs.
- In Section 3 we introduce some general-purpose graph operations and demonstrate the distinctive programming style of inductive graphs by discussing their implementation.

The next group of sections describes several graph algorithms and applications. These serve two purposes: first, they can be used as a reference to functions needed for particular problems, and second, they provide programming examples in FGL that help the user to better understand the concept of inductive graphs and to apply it to new problems.

- Section 4: Several versions of depth-first search and also derived operations, such as topological sorting and strongly connected components.
- Section 5: Breadth-first search and shortest paths (measured in number of edges).
- Section 6: Shortest paths (with respect to edge labels), here: Dijkstra's algorithm.
- Section 7: Operations to construct and query graph Voronoi Diagrams. An application is, for example, locating nearest facilities.
- Section 8: Minimum spanning trees, here: Prim's algorithm.
- Section 9: Maximum independent node sets.

Finally, in Appendix A we discuss the implementation of the graph type, and in Appendix B we provide some bibliographic background.

2 Inductive Graphs

The following definitions are contained in the module Graph. We describe the FGL graph types and their operations in several steps: Section 2.1 introduces the basic types. In Section 2.2 we demonstrate how graphs can be constructed, whereas in Section 2.3 we show functions to access information stored in a graph. Finally, Section 2.4 is describes elementary functions that enable the inductive decomposition of graphs.

Graph.hs

2.1 Graph Types

We define one type for directed node- and edge-labeled multi-graphs. This type captures the most general class of graphs. However, other graph types can be obtained as special cases: for example, undirected graphs can be well simulated by directed graphs having a symmetric edge structure, where we say that a directed graph g *properly represents* an undirected graph if for each edge (v,w,l) (that is, for each edge from node v to node w with label l) in g there is also an edge (w,v,l) in g. Moreover, unlabeled graphs simply have the node and/or edge label type () (unit/trivial type).

Plain nodes are represented by integers, and a plain edge is given by a pair of nodes. In the following we use the convention to denote the type of node labels by the type variable a and the type of edge labels by the type variable b. In addition to the types for plain nodes and edges (Node and Edge) and labeled nodes and edges (LNode and LEdge), we also include for convenience types UNode and UEdge for "quasi"-unlabeled nodes and edges, that is, nodes and edges which are labeled with the unit value () of type (). Finally, we call lists of nodes also paths.

```
type  Node     = Int
type LNode a   = (Node,a)
type UNode     = LNode ()

type  Edge     = (Node,Node)
type LEdge b   = (Node,Node,b)
type UEdge     = LEdge ()

type Graph a b = -- abstract type
type UGraph    = Graph () ()

type  Path     = [Node]
type LPath a   = [LNode a]
type UPath     = [UNode]
```

The inductive view of graphs is captured in the following description: a graph is either the empty graph (that contains no nodes) or a graph extended by a new node v together with its label and with edges to v's successors and predecessors. Each edge information contains the successor/predecessor node itself and the label of the edge. This information about a one-step inductive graph extension is gathered in an own type Context, which is called *context* because the new node is brought into some context of the existing graph. A list of (successor or predecessor) nodes paired with corresponding edge labels is also called an

adjacency. Contexts are not only used to build graphs, they are, in particular, employed when decomposing graphs. Since decomposition can fail for several reasons, the corresponding functions and their types have to account for this. Therefore, we have a type abbreviation for `Maybe`-contexts and also a type `Decomp` capturing the complete result of a decomposition: a pair of a possible context and a remaining graph. We also include type `GDecomp` for "guaranteed" decompositions, that is, decompositions that are expected to always have a context. Moreover, it is convenient to have variants of some of these types for unlabeled graphs. (The details will explained in Section 2.4. We just wanted to include the type definition in this section for easier later reference.)

```
type Adj b        = [(b,Node)]
type Context a b  = (Adj b,Node,a,Adj b)
type MContext a b = Maybe (Context a b)
type Decomp a b   = (MContext a b,Graph a b)
type GDecomp a b  = (Context a b,Graph a b)

type UContext     = ([Node],Node,[Node])
type UDecomp      = (Maybe UContext,UGraph)
```

2.2 Constructing Graphs

We have two basic functions to build up graphs inductively: the function `empty` denotes the empty graph, and the function `embed` extends a graph by a context:

```
empty :: Graph a b
embed :: Context a b -> Graph a b -> Graph a b
```

It is an error to try to insert a node that is already contained in the graph or to refer in predecessor or successor lists to nodes that are not contained in the graph to be extended. We have also included versions of these two functions for unlabeled graphs:

```
emptyU :: UGraph
embedU :: UContext -> UGraph -> UGraph
```

Sometimes it is convenient to use `embed` in infix notation. We have therefore also included the following function definition:

```
infixr &
c & g = embed c g
```

With the two functions `empty` and `embed/&` we are able denote graphs by terms. Let us begin by giving terms for several very simply graphs. These definitions can also be found in the module `GraphData`. We are constructing graphs of type `Graph Char ()`, that is, nodes are labeled by characters and edges are not labeled (that is, they are all labeled by the null value `()` of the unit or trivial type `()`).

GraphData.hs

```
a    = ([],1,'a',[]::[((),Node)]) & empty   -- just a node
loop = ([],1,'a',[((),1)])        & empty   -- loop on single node
e    = ([((),1)],2,'b',[])        & a       -- just one edge a-->b
ab   = ([((),1)],2,'b',[((),1)])  & a       -- cycle of two nodes a<-->b
```

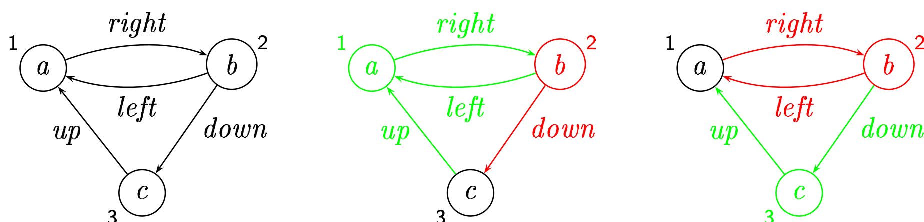

Figure 1: Directed graph with two inductive constructions.

To have a textual representation of graphs, we have defined the `show` function for graphs to print the adjacency list representation. This means that a graph is shown as a list of labeled nodes, each followed by the list of labeled outgoing edges. For example, the above graphs are printed as follows:

```
GraphData> a
1:'a'->[]

GraphData> loop
1:'a'->[((),1)]

GraphData> e
1:'a'->[((),2)]
2:'b'->[]

GraphData> ab
1:'a'->[((),2)]
2:'b'->[((),1)]
```

For a slightly larger example, consider the graph shown (on the left) in Figure 1.

We can build this graph of type `Graph Char String`, for example, with the following expression (see Figure 1, picture in the middle):

```
g3 = ([("left",2),("up",3)],1,'a',[("right",2)]) &
                        ([],2,'b',[("down",3)])  &
                        ([],3,'c',[])            & empty
```

The chosen order of inserting node contexts is not the only possible one. For example, we can also reverse the order. Then, however, the contexts have to be changed accordingly since we can refer in predecessor and successor lists only to nodes that are present in the graph to be extended. Note that consistency checks for graph construction are integrated into the definition of `embed` (and `&`). In fact, an error is reported when a context is added for a node that is already present in the graph or when a node mentioned in the successor or predecessor list is missing in the graph. Now the alternative construction of the graph is (see Figure 1, picture on the right):

```
g3' = ([("down",2)], 3,'c',[("up",1)])    &
      ([("right",1)],2,'b',[("left",1)]) &
                 ([],1,'a',[])           & empty
```

Again, this is only one further example, and, in fact, we can choose an arbitrary order of node insertion for building a graph. The textual representation of the graph `g3` (or `g3'`) is:

```
GraphData> g3
1:'a'->[("right",2)]
2:'b'->[("left",1),("down",3)]
3:'c'->[("up",1)]
```

`Graph.hs` In the module `Graph`, there are several additional functions defined that make the graph construction more convenient: the functions `insNode` and `insNodes` can be used to just insert one or more labeled nodes into a graph. The function `newNodes` can be helpful in cases a graph has to be extended whose construction is not known, that is, it is not clear which nodes are contained in it. Then the function `newNodes` can be used to get a sequence of nodes that are *not* contained in a graph. The functions `insEdge` and `insEdges` can be used to extend a graph by one or more labeled edges, and finally, the functions `mkGraph` and `mkUGraph` are very useful in building graphs in the traditional way, that is, by simply giving a list of nodes end edges. This, in particular, saves from inventing a sequence of contexts to denote a graph.

```
insNode  ::  LNode a  -> Graph a b -> Graph a b
insNodes :: [LNode a] -> Graph a b -> Graph a b
insEdge  ::  LEdge b  -> Graph a b -> Graph a b
insEdges :: [LEdge b] -> Graph a b -> Graph a b
newNodes :: Int -> Graph a b -> [Node]
mkGraph  :: [LNode a] -> [LEdge b] -> Graph a b
mkUGraph :: [Node]    -> [Edge]    -> UGraph
```

Several examples for the use of these functions can be found in the module `GraphData`. The functions `mkGraph` and `mkUGraph` can be useful in directly constructing graphs as well as in functions constructing specific graphs, such as `ucycle`, which constructs a cycle of specified length, and `star`, which builds a star-shaped graph:

```
ucycle :: Int -> UGraph
ucycle n = mkUGraph vs (map (\v->(v,v `mod` n+1)) vs)
           where vs = [1..n]

star :: Int -> UGraph
star n = mkUGraph [1..n] (map (\v->(1,v)) [2..n])
```

2.3 Extracting Graph Information

Now that we can build graphs, we next describe how to extract information from graphs. In this subsection we present some elementary graph access operations, whereas in the next subsection we introduce functions for a systematic graph exploration.

First, we have operations that deliver global information about a graph. For example, we can test with the function `isEmpty` whether a graph is empty or not. Moreover, we can count the number of nodes in a graph with `noNodes`, and we can extract the list of plain and labeled nodes from a graph with the functions `nodes` and `labNodes`. Similarly, the functions `edges` and `labEdges` compute the list of plain and labeled edges of a graph.

```
 isEmpty :: Graph a b -> Bool
 noNodes :: Graph a b -> Int
   nodes :: Graph a b -> [Node]
labNodes :: Graph a b -> [LNode a]
   edges :: Graph a b -> [Edge]
labEdges :: Graph a b -> [LEdge b]
```

Second, we can extract information about individual nodes: for a graph `g` that contains a node `v`, we can determine `v`'s successors (`suc`), predecessors (`pre`), or all neighbors (`neighbors`), the outgoing (`out`) or incoming edges (`inn`), and its different kinds of degrees (`outdeg`, `indeg`, and `deg`).

```
suc       :: Graph a b -> Node -> [Node]
pre       :: Graph a b -> Node -> [Node]
neighbors :: Graph a b -> Node -> [Node]
out       :: Graph a b -> Node -> [LEdge b]
inn       :: Graph a b -> Node -> [LEdge b]
outdeg    :: Graph a b -> Node -> Int
indeg     :: Graph a b -> Node -> Int
deg       :: Graph a b -> Node -> Int
```

Especially in graph traversals one frequently needs to access the information provided by the above functions for a particular, that is, already given, context. Therefore, we also provide the corresponding functions defined on contexts. In addition, we can extract the node and its label (or both) from a context.

```
suc'       :: Context a b -> [Node]
pre'       :: Context a b -> [Node]
neighbors' :: Context a b -> [Node]
out'       :: Context a b -> [LEdge b]
inn'       :: Context a b -> [LEdge b]
outdeg'    :: Context a b -> Int
indeg'     :: Context a b -> Int
deg'       :: Context a b -> Int
node'      :: Context a b -> Node
lab'       :: Context a b -> a
labNode'   :: Context a b -> LNode a
```

2.4 Graph Decomposition

The fundamental operation for decomposing inductive graphs is given by the function `match` that tries to locate a node `v` in a graph `g`, and yields `v`'s context, that is, its predecessors, its successors, and its label, and the remaining graph, that is, `g` without `v` and its incident edges. If `v` is not contained in `g`, no context is returned (represented by the constructor `Nothing`), and the returned graph is `g` itself.

```
match :: Node -> Graph a b -> Decomp a b
```

One can regard `match` as the (deterministic) inverse of `embed`. Again we have a special version for unlabeled graphs:

```
matchU :: Node -> UGraph -> UDecomp
```

Let us show some examples illustrating the meaning of `match`.

```
GraphData> match 1 a
(Just ([],1,'a',[]),)              -- remaining graph is empty

GraphData> match 1 loop
(Just ([],1,'a',[((),1)]),)        -- remaining graph is empty

GraphData> match 1 ab
(Just ([((),2)],1,'a',[((),2)]),
2:'b'->[])

GraphData> match 1 e
(Just ([],1,'a',[((),2)]),
2:'b'->[])

GraphData> match 1 g3
(Just ([("left",2),("up",3)],1,'a',[("right",2)]),
2:'b'->[("down",3)]
3:'c'->[])

GraphData> match 2 a
(Nothing,
1:'a'->[])
```

Look at the result of `match 1 loop`. We observe that the loop on node `1` was interpreted by `match` as an outgoing edge because in the resulting context the node `1` appears in the successor list. This is an arbitrary decision in the implementations of `match`, that is, we could as well have put loops in the predecessor list. However, we believe it would be an error to put loops into both lists at the same time because this would somewhat illegally augment the information obtained from the graph. This can be seen as follows: suppose `match` delivered loops in the successor and the predecessor lists. Then `match 1 a` would yield `(Just c,g)` as a result where `g` is the empty graph and `c` is the context

```
([((),1)],1,'a',[((),1)])   -- match loops in pre and suc list
```

Now, re-applying `embed`, that is, the expression `c & g`, does *not* yield the original graph `a`, but a graph with *two* loops on the node `1`:

```
1:'a'->[((),1),((),1)]      -- c & g
```

On the other hand with the chosen definition for `match`, we have the following nice property:

$$\texttt{match v g} = \texttt{(Just c,g')} \quad \Rightarrow \quad \texttt{c \& g'} = \texttt{g}$$

Since some applications might prefer loops delivered in predecessor lists, we also include a function `matchP` that does exactly this and behaves otherwise like `match`. In addition, there are also functions to match an arbitrary node or nodes with a specific property: `matchAny` matches an arbitrary node in a non-empty graph. Applied to an empty graph, `matchAny` produces an error. `matchAny` can be viewed as the non-deterministic inverse of `embed`, and it has the following property:

$$\texttt{isEmpty g} = \texttt{False} \quad \Rightarrow \quad \texttt{uncurry embed (matchAny g)} = \texttt{g}$$

Similarly, `matchSome` matches an arbitrary node that fulfills a condition. Again it is an error to apply `matchSome` to an empty graph, and an error is also produced when no node satisfies the given condition. Finally, `matchThe` matches a unique node of a certain property, that is, it matches whenever `matchSome` would match and the list of qualifying nodes contains exactly one node. We also have two functions `context` and `contextP` that match a context in a graph much like `match` and `matchP`, but return only the context and not the remaining graph. Dually, we also have a functions `delNode` and `delNodes` matches one or more nodes in the graph and ignore the resulting contexts and just yield the remaining graph. This amounts, in fact, to remove nodes from a graph. For completeness we also have included the corresponding functions `delEdge` and `delEdges` for removing one or more edges from a graph.

```
matchP    :: Node -> Graph a b -> Decomp a b
matchAny  :: Graph a b -> GDecomp a b
matchSome :: (Graph a b -> Node -> Bool) -> Graph a b -> GDecomp a b
matchThe  :: (Graph a b -> Node -> Bool) -> Graph a b -> GDecomp a b
context   ::  Node  -> Graph a b -> Context a b
contextP  ::  Node  -> Graph a b -> Context a b
delNode   ::  Node  -> Graph a b -> Graph a b
delNodes  :: [Node] -> Graph a b -> Graph a b
delEdge   ::  Edge  -> Graph a b -> Graph a b
delEdges  :: [Edge] -> Graph a b -> Graph a b
```

3 Basic Graph Operations and Their Implementation

In this Section we demonstrate the use of graph decomposition functions in the realization of some basic graph algorithms. The functions defined in the following are included in the module `Basic`. We start in Section 3.1 by briefly describing a set of basic graph operations that can be used for many everyday's tasks on graphs. In Section 3.2 we describe the implementation of these operators. This helps to understand the operations better and serves at the same time as a little tutorial on how to write recursive graph functions.

`Basic.hs`

3.1 General-Purpose Operations

The function `gmap` applies a function to all contexts of a graph; two specialized versions of `gmap` are the functions `nmap` and `emap` that map functions to node and edge labels. The function `grev` performs graph reversal, that is, `grev` swaps the direction of all edges in the graph. The function `undir` converts a directed graph into an undirected graph, that is, it adds edges to the graph so that it properly represents (see beginning of Section 2.1) an undirected graph. The function `unlab` forgets all labels of a graph and thereby converts it into an unlabeled graph, and `gsel` selects a list of contexts that satisfy a given property.

```
gmap  :: (Context a b -> Context c d) -> Graph a b -> Graph c d
nmap  :: (a -> c) -> Graph a b -> Graph c b
emap  :: (b -> c) -> Graph a b -> Graph a c
grev  :: Graph a b -> Graph a b
undir :: Graph a b -> Graph a b
unlab :: Graph a b -> UGraph
gsel  :: (Context a b -> Bool) -> Graph a b -> [Context a b]
```

In addition to these specific function we have also two very general fold operations on graphs: `ufold` successively decomposes all contexts from a graph and combines them in a right-associative way with a binary function of type `(Context a b) -> c -> c` and a unit value of type `c` into one value of type `c`. `ufold` is very similar to the well-known list fold, but an important difference is that the contexts are decomposed from the graph in an arbitrary order; therefore the "`u`" in the name that stands for *unordered.*

```
ufold :: ((Context a b) -> c -> c) -> c -> Graph a b -> c
```

The second fold function, `gfold`, decomposes graphs in a specific order. In particular, contexts are decomposed at specific positions, that is, nodes, in the graph. It takes essentially three function parameters to control the decomposition.

```
type Dir a b      = (Context a b) -> [Node]  -- direction of fold
type Dagg a b c d = (Context a b) -> c -> d  -- depth aggregation
type Bagg d c     = (Maybe d -> c -> c,c)    -- breadth/level aggregation

gfold :: (Dir a b) -> (Dagg a b c d) -> (Bagg d c) -> [Node] -> Graph a b -> c
```

Assume we have explored a graph to some part, that is, we are currently visiting a context `c` = `(p,v,l,s)`, and we have to explore the remaining graph `g`. Then `gfold` works as follows:

- The first function, say `f`, of type `(Context a b) -> [Node]` is applied to `c` and yields a list of nodes that are to be visited next. A typical example is the function `suc'` that yields the successors `s`; this specifies a depth-first traversal.

- The graph `g` is folded recursively at those nodes delivered by `f`. This yields a list of results `rs` which will be aggregated by the third parameter into a value `x` of type `c`. Now the second parameter function combines the current context and `x` into a value `y` of type `d`. Since values computed on deeper levels are combined with the current level, this process is also called *depth aggregation*.

- Each element of the list `rs` is a `Maybe d`-value resulting from the recursive graph fold of a node delivered by `f`: when the matching of the node is successful, it will be a value `Just y`, otherwise it will be the value `Nothing`. Now `rs` is list-folded by a function `g` and a unit value `u` (of type `c`) into a value of type `c`. Since values computed on the same level are combined, we call this process also *breadth aggregation*. Note that the third parameter is given by the pair `(g,u)`.

On the top level, `gfold` is applied to a list of nodes and yields a list of `Maybe d`-values. These are finally breadth-aggregated into a single value of type `c`.

As an example for the application of `gfold` consider the computation of a depth-first spanning tree. The direction of `gfold` is given by `suc'`, and the depth aggregation is `Br . node'` which selects the node of the current context and puts it at the root of a tree that has the recursively computed trees as subtrees. (The constructor `Br` is explained below in Section 4.) The breadth aggregation is given by the pair `(catMaybes,[])`:

```
dff :: [Node] -> Graph a b -> [Tree Node]
dff = gfold suc' (Br . node') (catMaybes,[])
```

3.2 Implementation of Graph Operations

In this section we discuss the implementation of some operations introduced above. We begin with the definition of `gmap`: if the graph is empty, the empty graph is the result, otherwise, we retrieve a context, apply `f` to it and embed the new context into the result of mapping `f` to the remaining graph. Note that by using `matchAny` to retrieve a context the order of encountering the contexts is arbitrary.

```
gmap :: (Context a b -> Context c b) -> Graph a b -> Graph c b
gmap f g | isEmpty g = empty
         | otherwise = f c & gmap f g'
         where (c,g') = matchAny g
```

An example for the application of `gmap` is setting all node labels to a character that corresponds to the node value:

```
gmap (\(p,v,_,s)->(p,v,chr(96+v),s))
```

Another example for a simple recursive graph function is the definition of `grev`. The structure of this definition is very similar to that of `gmap`: in the base case we return the empty graph, and if there is a context available, we swap successors and predecessors and embed this changed context in the reversal of the remaining graph. Again the chosen order of context decomposition does not matter.

```
grev  :: Graph a b -> Graph a b
grev g | isEmpty g = empty
       | otherwise = (s,v,l,p) & grev g'
       where ((p,v,l,s),g') = matchAny g
```

It is not difficult to see that `grev` can, in fact, be expressed as an instance of `gmap`:

```
grev = gmap (\(p,v,l,s)->(s,v,l,p))
```

Even gmap can be generalized; we can define it as an instance of ufold. Moreover, the functions gsel, nodes, undir and unlab can also be realized through ufold and gmap.

```
gmap f = ufold (\c->(f c&)) empty
gsel p = ufold (\c cs->if p c then c:cs else cs) []
nodes  = ufold (\(p,v,l,s)->(v:)) []
undir  = gmap (\(p,v,l,s)->let ps=nubBy (\x y->snd x==snd y) (p++s) in (ps,v,l,ps))
unlab  = gmap (\(p,v,_,s)->(unlabAdj p,v,(),unlabAdj s))
         where unlabAdj = map (\(_,v)->((),v))
```

4 Depth First Search

Depth-first search is one of the most basic and most important graph algorithms. It can reveal a lot about the internal structure of a graph, and this information can be used to implement several other algorithms, such as topological sorting or computing strongly connected components. The functions presented in this Section can be found in the module DFS. The definition of the tree data type is given in the module RoseTree.

DFS.hs

RoseTree.hs

A depth-first walk through a graph essentially means to visit each node in the graph once by visiting successors before siblings. The parameters of depth-first search are, of course, the graph to be searched, but in addition, a list of nodes is used saying which nodes are left to be visited. This list is needed for unconnected graphs where, after having completely explored one component, a node of another component is needed to continue the search. Depth-first search can deliver different kinds of results: the function dfs yields the list of nodes in the order visited (this list said to be in depth-first order). In contrast, the function dff computes a depth-first spanning tree, which keeps the edges that have been traversed to reach all the nodes. In fact, the result might not be a proper tree at all, which is the case when the graph is not connected. Therefore, dff actually delivers a list of trees, which is also called a *forest* (this also explains the use of a trailing f instead of a t in the name of the function).

The data type Tree and functions for computing pre- and postorder list of nodes are defined in the module RoseTree:

```
data Tree a = Br a [Tree a]

preorder   :: Tree a -> [a]
postorder  :: Tree a -> [a]
preorderF  :: [Tree a] -> [a]
postorderF :: [Tree a] -> [a]
```

The *structure* of the returned results (list or tree) is not the only possible variation of the depth-first search function. In fact, there are three additional dimensions of possible variation and generalization: (i) *direction*: we can also follow predecessors in addition or instead of successors, (ii) *result*: we can return other data than just the visited nodes, and (iii) *start*: we can fix the initial list of nodes to be the nodes of the graph to be traversed. Altogether

we obtain 32 variations of depth-first search. The names of all these functions are given by the following small grammar:

dfs-fun	→	[*dir*]`df`*struct*[`With`][`'`]
dir	→	`x` | `u` | `r`
struct	→	`s` | `f`

The meaning of any such function can be derived from the effect of the given parameters:

dir	
`x`	The functions are parameterized by a function of type `Context a b -> [Node]` that determines in each step the list of nodes to be visited next. For plain `dfs` this is just the function `suc'`.
`u`	The nodes to be visited next are successors and predecessors. This means that these versions of depth-first search ignore the direction of the edges and are therefore also called *unordered* depth-first search. `udfs` can be expressed as an instance of `xdfs` with the direction parameter `neighbors'`.
`r`	The nodes to be visited next are the predecessors instead of the successors. Thus, these depth-first search functions move in the opposite direction, which has the same effect as performing "normal" depth-first search on the reversed graph. We call this option *reverse* depth-first search, and `rdfs` can be expressed as an instance of `xdfs` with the direction parameter `pre'`.
␣	(No Prefix) This is the default case in which functions just follow successors. `dfs` can be expressed as an instance of `xdfs` with the direction parameter `suc'`.
struct	
`s`	The result of these functions are lists: for the "`With`" versions the list elements are computed from the contexts by a parameter function, and for the "normal" versions, these are just nodes. In other words, `dfs` can be expressed as `dfsWith nodes'`.
`f`	The result of these functions are trees (or, more precisely, forests). Again, the "`With`" functions extract the elements put into the trees from the contexts by a parameter function, whereas the "normal" functions simply take the current node. This means, `dff` can be expressed as `dffWith nodes'`.
`With`	The objects to be put into lists or trees are computed from the visited contexts by a parameter function of type `Context a b -> c` (abbreviated as the type `CFun`). Function names without the "`With`" suffix just take the nodes from the context.
`'`	To ensure that a graph is always completely explored (without caring for its connectedness) we can call functions like `dfs` just with the list of all nodes of the graph. Therefore we have included primed versions of depth-first search functions that do not have a node list as a parameter and instead take the list of nodes of their graph argument. In the default case, node lists have to be provided explicitly.

We have not included all 32 functions in the library, but only the ones that seem to be most useful.

```
type CFun a b c = Context a b -> c

xdfsWith  :: CFun a b [Node] -> CFun a b c -> [Node] -> Graph a b -> [c]
xdffWith  :: CFun a b [Node] -> CFun a b c -> [Node] -> Graph a b -> [Tree c]
 dfsWith  :: CFun a b c -> [Node] -> Graph a b -> [c]
 dffWith  :: CFun a b c -> [Node] -> Graph a b -> [Tree c]
 dfsWith' :: CFun a b c -> Graph a b -> [c]
 dffWith' :: CFun a b c -> Graph a b -> [Tree c]
 dfs      :: [Node] -> Graph a b -> [Node]
 dff      :: [Node] -> Graph a b -> [Tree Node]
 dfs'     :: Graph a b -> [Node]
 dff'     :: Graph a b -> [Tree Node]

udfs      :: [Node] -> Graph a b -> [Node]
udff      :: [Node] -> Graph a b -> [Tree Node]
rdfs      :: [Node] -> Graph a b -> [Node]
rdff      :: [Node] -> Graph a b -> [Tree Node]
udfs'     :: Graph a b -> [Node]
udff'     :: Graph a b -> [Tree Node]
rdfs'     :: Graph a b -> [Node]
rdff'     :: Graph a b -> [Tree Node]
```

There are two ways of generalizing `xdfsWith` and `xdffWith` further: first, we can abstract from the type constructor (list or forest) that capture the resulting values, and second, the intermediate results computed for all, say, successors, of a node are currently collected in a list, and this can be generalized into an arbitrary function to combine the values. Both of these generalizations lead directly to the definition of the function `gfold`, see Section 3. Hence, all the shown depth-first search functions (and more) can be expressed as an instance of `gfold`.

Finally, there are several graph problems that can be solved with the help of the above depth-first search functions: `components` computes the (simply) connected components as a list of node lists, `noComponents` says how many components a graph consists of. If this number is 1, `isConnected` yields `True`. The function `topsort` computes a topologically sorted list of nodes (for acyclic graphs), `scc` computes the strongly connected components, and the function `reachable` determines all nodes that are reachable in a graph from a specific node.

```
components   :: Graph a b -> [[Node]]
noComponents :: Graph a b -> Int
isConnected  :: Graph a b -> Bool
topsort      :: Graph a b -> [Node]
scc          :: Graph a b -> [[Node]]
reachable    :: Node -> Graph a b -> [Node]
```

5 Breadth-First Search

Breadth-first search essentially means visiting siblings before successors. This has the effect to first visit all nodes of a certain distance (measured in number of edges) from the start node before visiting nodes that are further away. This property is exploited by the shortest path function `esp` given below that is based on breadth-first search. This and all other functions can be found in the module `BFS`. `BFS.hs`

Similar to depth-first search we have variations of breadth-first search along different dimensions. Usually, a breadth-first search takes a single node as a parameter, and then explores the graph. In some applications, however, (for example, locating nearest facilities [10]) it is required that the search starts from several nodes, so to say "in parallel". Functions that have a suffix "`n`" account for this and take as a parameter a list of start nodes. The second dimension is, again, the use of "`With`" suffixes to enable to feed a parameter function that extracts the interesting information from the visited contexts.

Now we have four versions of breadth-first search for computing just nodes (for example, `bfs`) or other information (for example, `bfsWith`) and starting from a single node or from a list of nodes (for example, `bfsn`). The functions `level` and `leveln` compute the breadth-first list in which the nodes are paired with their level, that is, with their distance to the start node(s). The functions `bfe` and `bfen` return the list of traversed edges of a breadth-first search. Note that `bfen` requires a list of edges whereas `bfe` just takes one node. Finally, the function `bft` computes a breadth-first spanning tree, and the function `esp` computes the shortest path between two nodes where the length of a path is given by the number of edge in it.

```
bfs      ::  Node  -> Graph a b -> [Node]
bfsn     :: [Node] -> Graph a b -> [Node]
bfsWith  :: (Context a b -> c) ->  Node  -> Graph a b -> [c]
bfsnWith :: (Context a b -> c) -> [Node] -> Graph a b -> [c]

level    ::  Node  -> Graph a b -> [(Node,Int)]
leveln   :: [(Node,Int)] -> Graph a b -> [(Node,Int)]
bfe      ::  Node  -> Graph a b -> [Edge]
bfen     :: [Edge] -> Graph a b -> [Edge]

bft      :: Node -> Graph a b -> RTree
esp      :: Node -> Node -> Graph a b -> Path
```

It remains to explained how the type `RTree` is defined. Since the definition of `esp` is based on such a tree (computed by `bft`) we consider in some detail how shortest paths can be computed to motivate the definition of `RTree`.

To build a breadth-first spanning tree we have to keep more information than just the order of nodes. Before we present an algorithm for this we make two observations: first, it is quite difficult to efficiently build a breadth-first spanning tree represented, for example, as a `Tree Node` value as was done, for example, by `dff`. The problem is that the expressions denoting such trees have to be built bottom-up whereas the recursion in `bfs` delivers nodes in a way that is per se suited for top-down construction. Second, such a representation is

not so important anyhow because finding a shortest path with the help of a breadth-first spanning tree is supported by *inward* directed trees, that is, trees whose edges point from the successors toward predecessors: finding a shortest path from node s to node t can be achieved by (i) computing the breadth-first spanning tree rooted at s, (ii) locating node t in it, and (iii) following the edges from t to the root. Then the reverse list of traversed nodes/edges gives the shortest path.

Now an inward directed tree can be represented simply as a mapping with domain and range of type `Node` mapping nodes to their predecessors. Since such a mapping is built incrementally either during breadth-first search or after it using a list of traversed edges, we cannot use a monolithic array for implementing it. In fact, the array construct proposed in [13] could be used to build up such a tree, but this requires to write the whole algorithm serving the array construction, and this destroys the simplicity and elegance of the functional `bfs` algorithm. Instead we can use a binary search tree, but this adds a logarithmic factor on each operation for (i) building up the spanning tree and (ii) for reconstructing the shortest path after that.

The latter problem can be addressed by not just mapping nodes to their predecessor, but to the whole path to the root. This does not really make the implementation more complex: to insert u as a predecessor of v instead of just inserting u with key v into the tree, we first locate the root path already stored at u, say p, and then insert u:p with key v into the tree. In this way we only need to locate t in the inward directed tree, and we can just reverse the list of stored nodes to obtain the shortest path from s to t. Note that this representation causes only minimal space overhead: since common prefixes of paths are shared, this representation is linear in the number of stored nodes. However, still the complexity of computing the breadth-first spanning tree and thus also for computing shortest paths is $O(n \log n + e)$.

Now a further improvement is to represent a breadth-first spanning tree by a list (instead of a tree) of paths from each node to the root. We call these kind of trees *root path trees*. Again, to have a linear space requirement the paths should share common prefixes. This can be achieved quite easily by keeping these paths in the queue used by breadth-first search.

Now the algorithm for finding the shortest path between two nodes s and t first computes the breadth-first spanning tree rooted at s. This spanning tree is represented as a list of root paths, and from these the first one that has t as a first element is extracted. This root path has then only to be reversed to obtain the shortest path.

Hence, we have tho following definitions for root path trees, which can be found in the module `RootPath`. The type `LRTree` is the type of *labeled root path trees*, these are root path trees in which all nodes of all root paths are labeled. These kinds of trees will be needed in Section 6. The function `getPath` finds a path from the root of the tree to the given node, and the function `getLPath` finds a labeled path in a labeled root path tree. The function `getLabel` yields the distance of a node (measured in accumulated edge costs) from the root.

`RootPath.hs`

```
type RTree    = [Path]
type LRTree a = [LPath a]

getPath     :: Node ->  RTree   -> Path
getLPath    :: Node -> LRTree a -> LPath a
getDistance :: Node -> LRTree a -> a
```

6 Shortest Paths

Very closely related to the shortest path algorithm of the preceding section is Dijkstra's algorithm for computing shortest paths in graphs with positive edge labels. It is defined in the module `SP`. The main difference is that the length of a path is now defined to be the sum of its edge labels and that a shortest path between two nodes is accordingly one that has a minimum path length. The function `spTree` computes a shortest path tree, and the function `sp` determines a shortest path between two nodes. This is an unlabeled path, and to obtain the length of a shortest path the function `spLength` can be used.

`SP.hs`

```
spTree   :: Real b => Node -> Graph a b -> LRTree b
spLength :: Real b => Node -> Node -> Graph a b -> b
sp       :: Real b => Node -> Node -> Graph a b -> Path
```

We have not included a possible generalization of these functions that take as an additional parameter a function, say, `f`, on edge labels that delivers values according to which shortest paths are to be determined because the same behavior can be achieved by first mapping `f` to all edges (with `emap`) and then using the above functions.

7 Graph Voronoi Diagram

The *Voronoi Diagram* with respect to a set of points $K = \{v_1, \ldots, v_k\}$ (in the Euclidean plane), called *Voronoi points*, is a subdivision of the plane into regions $R_1, \ldots, R_k$, called *Voronoi regions*, such that for any point $p \in R_i$ the distance to v_i is not larger than to any other $v \in K$. The concept of Voronoi Diagram extends straightforwardly to graphs as follows [10]: K is given by a set of nodes and determines a set of *Voronoi sets* $\{N_0, \ldots, N_k\}$ such that (i) the shortest path from a node $v \in N_i$ to $v_i \in K$ is not longer than to any other node in K and (ii) N_0 contains all nodes for which all nodes K are unreachable. The node partition $\{N_0, \ldots, N_k\}$ is called the *(inward) graph Voronoi Diagram*. This means that the *inward* Voronoi diagram is based on the paths which lead *toward* the Voronoi nodes, and there is a dual definition of the *outward* graph Voronoi Diagram that is based on shortest paths *from* Voronoi nodes to nodes of the Voronoi sets.

One important application of graph Voronoi diagrams is the location of nearest facilities. Assume that the graph models a transportation network and that K is a set of nodes where facilities, such as, fire stations, hospitals, post offices, or shopping malls, are located. Now the *nearest facility* of a query point v is that Voronoi node to which the shortest path from v has minimal cost. The following information may be of interest: "Which Voronoi node v_i is nearest to v ?", "How far is v_i from v ?", or "What is the shortest path from v to v_i ?". As an example suppose an accident happens at node v. Then we seek the shortest path to the nearest hospital.

We can reuse the type `LRTree` to represent graph Voronoi Diagrams. The difference to a shortest path tree is that it has, in general, more than one root, in other words, it is actually a kind of *shortest path forest*. The operations provided by module `GVD` are the following: the function `gvdIn` computes the inward graph Voronoi Diagram, whereas the function `gvdOut` computes the outward graph Voronoi Diagram. The function `nearestNode` computes the

`GVD.hs`

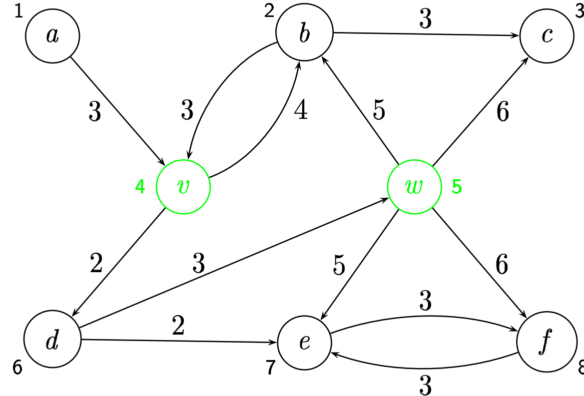

Figure 2: Directed graph with Voronoi nodes displayed in green.

nearest Voronoi node for a given node, the function **nearestDistance** yields the length of the shortest path to the nearest Voronoi node (for inward Voronoi Diagrams) and the length of the shortest path from the nearest Voronoi node (for outward Voronoi Diagrams), and the function **nearestPath** returns the shortest path to the nearest Voronoi node (for inward Voronoi Diagrams) and the shortest path from the nearest Voronoi node (for outward Voronoi Diagrams).

```
type Voronoi a = LRTree a

gvdIn       :: Real b => [Node] -> Graph a b -> Voronoi b
gvdOut      :: Real b => [Node] -> Graph a b -> Voronoi b
voronoiSet  :: Real b => Node -> Voronoi b -> [Node]
nearestNode :: Real b => Node -> Voronoi b -> Maybe Node
nearestDist :: Real b => Node -> Voronoi b -> Maybe b
nearestPath :: Real b => Node -> Voronoi b -> Maybe Path
```

As an example consider the graph `vor` shown in Figure 2. When computing the inward and outward graph Voronoi Diagram for the Voronoi nodes 4 and 5, we get, for example:

```
GVD> voronoiSet 4 (gvdIn [4,5] vor)
[4,1,2]

GVD> voronoiSet 5 (gvdIn [4,5] vor)
[5,6]

GVD> voronoiSet 4 (gvdOut [4,5] vor)
[4,6,7,2]
```

8 Minimum Spanning Tree

A minimum spanning tree is a spanning tree of a labeled undirected graph of minimal total edge length. Hence, in our context of directed graphs the presented functions work, in

general, only for directed graphs that properly represent undirected graphs. Remember that we can easily convert any directed graph into one representing an undirected one with the function `undir` described in Section 3.1.

We have to decide about the representation of the spanning tree, and this decision depends on the context in which the spanning tree is used. One application can be found in telecommunication: some telephone companies calculate the costs of phone calls by the length of a path between two nodes in a precomputed minimum spanning tree. This is supported again by root path trees. Now the following functions are available in the module `MST`: the function `msTreeAt` for computing a minimum spanning tree that is rooted at a particular node and the function `msTree` that computes a minimum spanning tree for an arbitrarily chosen root. Finally, the function `msPath` finds a path between two nodes in a minimum spanning tree.

MST.hs

```
msTreeAt :: Real b => Node -> Graph a b -> LRTree b
msTree   :: Real b => Graph a b -> LRTree b
msPath   :: Real b => LRTree b -> Node -> Node -> Path
```

9 Maximum Independent Node Sets

An independent node set is a subset of the nodes of a graph such that no two nodes of this set are connected by an edge. A maximum independent node set is an independent node set of maximum cardinality. The maximum independent node set is in a sense the dual of the maximum clique problem which asks for a maximal set of nodes such that each pair of nodes is connected by an edge. This problem was included here because the implementation makes use of that fact that inductive graphs are persistent data structures [18].

In the module `Indep` we have just one function `indep` to compute for a graph a maximum independent node set.

Indep.hs

```
indep :: Graph a b -> [Node]
```

References

[1] A. Aasa, S. Holström, and C. Nilsson. An Efficiency Comparison of Some Representations of Purely Functional Arrays. *BIT*, 28(3):490–503, 1988.

[2] P. Briggs and L. Torczon. An Efficient Representation For Sparse Sets. *ACM Letters on Programming Languages*, 2(4):59–69, 1993.

[3] F. W. Burton and H.-K. Yang. Manipulating Multilinked Data Structures in a Pure Functional Language. *Software – Practice and Experience*, 20(11):1167–1185, 1990.

[4] P. F. Dietz. Fully Persistent Arrays. In *Workshop on Algorithms and Data Structures*, LNCS 382, pages 67–74, 1989.

[5] M. Erwig. Graph Algorithms = Iteration + Data Structures? The Structure of Graph Algorithms and a Corresponding Style of Programming. In *18th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, pages 277–292, 1992.

[6] M. Erwig. Fully Persistent Graphs – Which One to Choose? In *9th Int. Workshop on Implementation of Functional Languages*, LNCS 1467, pages 123–140, 1997.

[7] M. Erwig. Functional Programming with Graphs. In *2nd ACM Int. Conf. on Functional Programming*, pages 52–65, 1997.

[8] M. Erwig. A Functional Homage to Graph Reduction. Technical Report 239, FernUniversität Hagen, 1998.

[9] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

[10] M. Erwig. The Graph Voronoi Diagram with Applications. *Networks*, 2000. To appear.

[11] L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *23rd ACM Symp. on Principles of Programming Languages*, pages 284–294, 1996.

[12] J. Gibbons. An Initial Algebra Approach to Directed Acyclic Graphs. In *Mathematics of Program Construction*, LNCS 947, pages 282–303, 1995.

[13] T. Johnsson. Efficient Graph Algorithms Using Lazy Monolithic Arrays. *Journal of Functional Programming*, 8(4):323–333, 1998.

[14] Y. Kashiwagi and D. Wise. Graph Algorithms in a Lazy Functional Programming Language. In *4th Int. Symp. on Lucid and Intensional Programing*, pages 35–46, 1991.

[15] D. J. King. *Functional Programming and Graph Algorithms.* PhD thesis, University of Glasgow, 1996.

[16] D. J. King and J. Launchbury. Structuring Depth-First Search Algorithms in Haskell. In *22nd ACM Symp. on Principles of Programming Languages*, pages 344–354, 1995.

[17] J. Launchbury. Graph Algorithms with a Functional Flavour. In *Advanced Functional Programming*, LNCS 925, pages 308–331, 1995.

[18] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK, 1998.

[19] M. E. O'Neill and F. W. Burton. A New Method for Functional Arrays. *Journal of Functional Programming*, 7(5):487–513, 1997.

[20] S. L. Peyton Jones, J. Hughes *et al.* Report on the Programming Language Haskell 98. 1999.

Appendix A: Implementation Aspects

The implementation of inductive graphs has to support the following operations for constructing and decomposing graphs:

Construction		**Decomposition**	
Empty graph	(`empty`)	Test for empty graph	(`isEmpty`)
Add context	(`embed`)	Extract arbitrary context	(`matchAny`)
		Extract specific context	(`match`)

In particular, graphs have to be fully persistent, that is, updates on a graph must leave previous versions intact.

Graph Representations and Persistence

One idea is to use a plain term representation. This is attractive because it offers persistence for free. However, a closer look rules out this option because the implementation of `match` is hopelessly inefficient, and even the implementation of `embed` is inefficient since it has to ensure the existence of the predecessors and successors and the non-existence of the newly inserted node, and testing this takes at least linear time with respect to the size of the graph.

Considering the imperative world, there are two main representations that both have their strengths and drawbacks: adjacency lists and incidence matrices. Except for special applications, the adjacency lists graph representation is generally favored over the incidence matrix because (i) its space requirement is linear in the graph size compared to $\Omega(n^2)$ space for the matrix and (ii) adjacency lists offer $O(1)$ access time to the successors of an arbitrary node in contrast to $\Omega(n)$ time needed to scan a complete row in an incidence matrix.

We have therefore concentrated on two alternatives for making adjacency lists persistent: the first representation uses a variant of the version tree implementation of functional arrays,[1] and the second representation stores successor and predecessor lists in a balanced binary search tree. The version tree implementation is based on the proposal [1] and records changes to the original array in an inward directed tree of (index, value) pairs that has the original array at its root.[2] Each different array version is represented by a pointer to a node in the version tree, and the nodes along the path to the root mask older definitions in the original array (and the tree). Adding a new node to the version tree can be done in constant time, but index access might take up to u steps where u denotes the number of updates to the array. We have extended this basic structure by an additional cache array and a further array carrying time stamps for nodes. Moreover, to support some specialized operations efficiently, this structure is supplemented by a two-array implementation of node partitions to keep track of inserted and deleted nodes.

[1] Note that the current version of FGL/Haskell does not contain the version tree implementation because it needs destructive array updates. In contrast, FGL/ML does contain a version tree implementation.

[2] In fact, there are more sophisticated functional array implementations available, for example, [4] and [19]. However, the implementation requires considerable effort, and benchmarks have shown that even the simpler version tree implementation does not hold in practice what its asymptotic complexity promises [6].

Optimizing the Version Tree Array Representation

In the version tree representation the implementation of `match` becomes quite inefficient since the deletion of a context `(p,v,l,s)` requires the removal of `v` from each of its predecessor's successor list and from each of its successor's predecessor list. When c denotes the size of the context ($c \approx$ `length p` $+$ `length s`), this means a runtime of $O(uc^2)$ (recall that u gives the total number of previous updates to the graph). By keeping the predecessors and successor in balanced binary search trees the effort can be reduced to $O(uc \log c)$.

Avoiding Node Deletion. To avoid these costly deletion activities we equip each node in the graph with a positive integer, and this integer is negated once the node is deleted. Positive node stamps are also put into successor/predecessor lists. Now when a node context is deleted, we need not remove `v` from all referencing successor and predecessor lists because when a successor list `l` (of a node `w`) is accessed that contains `v`, all elements that have non-matching stamps are ignored, that is, `v` will not be returned as a successor because it has a negative node stamp whereas `l` contains `v` with a positive stamp. When `v` is re-inserted into the graph later, we make the stamp of `v` positive again and increase it by 1, and we take this new stamp over to all newly added predecessors and successors. Now if `w` is not among the new predecessors, the old entry in `l` is still correctly ignored when `l` is accessed because its value smaller than `v`'s current stamp.

One might wonder whether the garbage nodes in successor and predecessor lists (that is, invalid and unused references to deleted nodes) are a source of inefficiency. In practice, this does not seem to be a problem. For example, in the case of graph reduction, where graphs are heavily updated, only 25-30% of nodes in successor and predecessor lists are filtered out due to invalid stamps.

Avoiding Version Tree Lookups. We add an imperative cache array to the leftmost node of the version tree. This means that the array represented by that node is, in fact, duplicated. Since index access within this array is possible in constant time, algorithms that use the functional array in a single-threaded way have the same complexity as in the imperative case, since the version tree degenerates to a left spine path with the leaf node offering constant time access during the whole algorithm.

There is a subtlety in this implementation having just one cache array: if a functional array is used a second time, the cache has already been consumed for the previous computation and cannot be used again. This gives a surprising time behavior: the user executes a program on a functional array, and it runs quite fast. However, running the same program again results, in general, in much larger execution times since all access now goes through the version tree. Therefore, we create in our implementation a new cache for each new version derived from the original array.

Support for Special Operations. The version tree implementation described so far is surprisingly inefficient for the operations `isEmpty` and `matchAny`. Testing for the empty graph can be easily supported by extending the graph representation to include the number of nodes in the graph. More difficulties presents the operation `matchAny` for which we have, in general, to scan the whole stamp array to find a valid, that is, non-deleted, node. Note

that even a simple imperative array implementation requires, in general, linear time for this operations by scanning the whole array. (This is not surprising since the question of graph updates is completely ignored anyway in almost all descriptions of imperative graph representations.)

To account for `matchAny` we keep for each graph a partition of inserted nodes (that is, nodes existent in the graph) and deleted nodes: when a node is deleted (decomposed), it is moved from the inserted-set into the deleted-set, when a node is inserted into the graph, it is moved the other way. The node partition is realized by two arrays, *index* and *elem*, and an integer k giving the number of existent nodes, or, equivalently, pointing to the last existing node. The array *elem* stores all existent nodes in its left part and all deleted nodes in its right part, and *index* gives for each node its position in the *elem* array. A node v is existent if $index[v] \leq k$, likewise, it is deleted if $index[v] > k$. Inserting a new node v means to move it from the deleted-set into the inserted-set. This is done by exchanging v's position in *elem* with the node stored at $elem[k+1]$ (that is, the first deleted node) followed by increasing k by 1. The entries in *index* must be updated accordingly. To delete node v, first swap v and $elem[k]$, and then decrease k by 1. All this is possible in constant time.

Now all the above mentioned graph operations can be implemented to work in constant time: `matchAny` can be realized by calling `match` with $elem[1]$, and `isEmpty` is true if $k = 0$. Moreover, some other useful graph operations are efficiently supported by the node partition: a list of i fresh nodes, that is, nodes that are not contained in a graph, is simply given by $[elem[k+1], \ldots, elem[k+i]]$ (this operation is needed, for example, to extend a graph whose construction history is not known), k gives the number of nodes in the graph, and all nodes can be reported in time $O(k)$ which might be much less than the size of the array. The described implementation of node partitions is an extension of the sparse set technique proposed in [2]. The drawback of the described extension is that keeping the partition information requires additional space and causes some overhead. Moreover, arrays do not become truly dynamic since they can neither grow nor shrink.

Binary Search Tree Representation

A binary search tree can be well used as a functional array implementation, and this offers an immediate realization of functional graphs: 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. Note that m is used to support the creation of new nodes. Keeping the largest node value used in the graph, this is possible in $O(1)$ time.

However, inserting and deleting a node context `(p,v,l,s)` requires considerable effort: concerning insertion we have to insert the context itself (which takes $O(\log n)$ steps when n nodes are in the graph), and we have to insert `v` as a successor (predecessor) for each node in `p` (`s`) (which takes $O(c \log n)$ steps). Hence, insertion runs in $O(c \log n)$ time which can be as large as $O(n \log n)$ for dense graphs. Context deletion takes even more time since we have to remove `v` from the successor (predecessor) list for each element of `p` (`s`), which requires searching these lists for `v`. Altogether deletion runs in $O(c^2 \log n)$ time or $O(c \log c \log n)$ if predecessors/successors are stored as search trees. In dense graphs, this gives a complexity of $O(n^2 \log n)$ and $O(n \log^2 n)$.

Although the asymptotic behavior of the search tree representation is clearly worse (at

least for single-threaded graph uses) than the array implementation, it performs very well in practice (see [6]), maybe because it is much simpler and does not require so many tunings. It also has the great advantage that it is a truly dynamic structure that supports unbounded growth of graphs. A further problem with the array implementation is that in order to be implemented in Haskell it needs to exploit unsafe features because operations like constant time array updates have to be encapsulated in a monad, and this monad has to extend as far as access to the array is made, that is, the monad would eventually show up in the algorithms and cannot be hidden in the graph implementation which is very bad and would completely destroy the functional flavor of the algorithms using inductive graphs. The version tree implementation is therefore only contained in the ML version of FGL, the Haskell version currently provides only the search tree representation.

Appendix B: Further Reading

There exist quite different proposals for how to implement graphs and graph algorithms in functional programming languages. Here we collect some links for further study of the topic.

A straightforward approach proposed in [3] is to pass the state used by graph algorithms through function calls where the state itself is represented by a functional array. This is certainly a standard way of implementing any imperative algorithm in a functional language. Burton and Yang show how classical algorithms can be translated into a lazy functional language, but no particular use of functional languages is made in the design of the algorithms themselves.

In contrast, in [14] algorithms are described as fixed points of recursive equations which essentially relies on lazy evaluation. This approach exploits and relies on features of lazy functional languages. However, the algorithms become quite complex and are rather difficult to comprehend. As with [3] this approach does not achieve the asymptotic runtime of imperative algorithms.

A kind of combinator approach was presented in [5]: we have identified some classes of graph algorithms and have introduced a few corresponding predefined operators. A graph algorithm is realized by selecting an operator and providing it with appropriate parameter functions and data structures. We believe that the approach reflects the structure of graph algorithms very well. However, like in the previous two approaches there is not much potential for formal program manipulation. Another drawback is that the combinator approach is limited in expressiveness.

The proposal of [16] is concerned only with depth-first search, and the focus is on a generated data structure, the depth-first spanning forest, instead of the underlying graph algorithm. This facilitates formal reasoning, in particular, the formal development of many algorithms based on depth-first search becomes possible. The depth-first search function itself is realized nicely in a generate-and-prune manner. Monads are used to implement the state maintained during the search (that is, the vertices visited) to achieve linear running time. At this point the approach is stuck with the imperative programming style. Although encapsulated and restricted to a single point, it comes up in the process of program fusion where transformations become quite complex when functions are moved across state transformers, see [17] where it is demonstrated how phase fusion can be applied to eliminate intermediate results of some of these algorithms. King [15] defines in his thesis many more

algorithms, but as with depth-first search, the defined functions are mainly implementations of imperative algorithms.

Fegaras and Sheard [11] investigate a generalization of fold operations to data types with embedded functions. As one motivating example they show how to model graphs. However, that approach is somewhat limited (it is not clear how to define, for example, a function for reversing all edges in a graph) and it is highly inefficient since direct access to a node requires, in general, traversal of the whole graph.

Also related is the work of Gibbons [12] who considers the definition of graph fold operations within an algebraic framework. But he deals only with acyclic graphs, and an implementation is not discussed, so that his approach is actually not usable.

In contrast to the monolithic view of graphs which is so dominating that it is even adopted by most functional approaches, we suggest to view graphs inductively, as a data type defined by two constructors, much like lists or trees. This view was first presented in [7] where the focus was to define several kinds of graph fold operations and to identify laws for them that can be used for program transformation. Also a first implementation of functional graphs was provided. In [6] we have extended the implementation in several ways and have compared different representation schemes by performing some benchmarks. The inductive graph view has also applications that go beyond the realization of functional graph algorithms, for example, inductive graphs have facilitated the denotational semantics definition of visual languages [9]. Another application, which has an educational pretension, is the purely functional description of graph reduction [8].

Index

`&` (function), 4

`Adj` (type), 4
adjacency, 4

`Basic` (module), 9
`bfe` (function), 15
`bfen` (function), 15
`BFS` (module), 15
`bfs` (function), 15
`bfsn` (function), 15
`bfsWith` (function), 15
`bft` (function), 15
binary search tree, 24
`Br` (constructor), 11, 12
breadth aggregation, 11
breadth-first search, 15–16
breadth-first spanning tree, 15

`CFun` (type), 13, 14
component
    connected, 14
    strongly connected, 14
`components` (function), 14
`Context` (type), 3
context, 3
`context` (function), 9
`contextP` (function), 9

`Decomp` (type), 4
`deg` (function), 7
`deg'` (function), 7
`delEdge` (function), 9
`delEdges` (function), 9
`delNode` (function), 9
`delNodes` (function), 9
depth aggregation, 10
depth-first forest, 12
depth-first order, 12
depth-first search, 12–14
    reverse, 13
    unordered, 13
depth-first spanning tree, 11, 12
`dff` (function), 11–15
`dff'` (function), 14
`dffWith` (function), 13, 14
`dffWith'` (function), 14
`DFS` (module), 12
`dfs` (function), 12–14
`dfs'` (function), 14
`dfsWith` (function), 13, 14
`dfsWith'` (function), 14
directed graph, 3

`Edge` (type), 3
edge, 3
`edges` (function), 6
`emap` (function), 17
`embed` (function), 4, 22
`empty` (function), 4, 22
`esp` (function), 15

forest, 12, 13
    depth-first, 12
    shortest path, 17

`GDecomp` (type), 4
`getLabel` (function), 16
`getLPath` (function), 16
`getPath` (function), 16
`gfold` (function), 10, 11, 14
`gmap` (function), 10–12
`Graph` (module), 3, 6
graph
    directed, 3
    labeled, 3
    undirected, 3, 18
    unlabeled, 3
graph Voronoi Diagram, 17
    inward, 17
    outward, 17
`GraphData` (module), 4, 6
`grev` (function), 10, 11
`gsel` (function), 10, 12
`GVD` (module), 17
`gvdIn` (function), 17
`gvdOut` (function), 17

indeg (function), 7
indeg' (function), 7
Indep (module), 19
indep (function), 19
independent node set, 19
inn (function), 7
inn' (function), 7
insEdge (function), 6
insEdges (function), 6
insNode (function), 6
insNodes (function), 6
inward directed tree, 16
inward graph Voronoi Diagram, 17
isConnected (function), 14
isEmpty (function), 6, 22–24

lab' (function), 7
labEdges (function), 6
labeled graph, 3
labeled root path tree, 16
labNode' (function), 7
labNodes (function), 6
LEdge (type), 3
level (function), 15
leveln (function), 15
LNode (type), 3
loop, 8
LPath (type), 3, 16
LRTree (type), 16, 17

match (function), 7, 22–24
matchAny (function), 9, 11, 12, 22–24
matchP (function), 9
matchSome (function), 9
matchThe (function), 9
maximum independent node set, 19
MContext (type), 4
minimum spanning tree, 18
mkGraph (function), 6
mkUGraph (function), 6
msPath (function), 19
MST (module), 19
msTree (function), 19
msTreeAt (function), 19
multi-graph, 3

nearest facility, 17
nearestDistance (function), 18
nearestNode (function), 17
nearestPath (function), 18
neighbors (function), 7
neighbors' (function), 7, 13
newNodes (function), 6
noComponents (function), 14
Node (type), 3
node, 3
node partition, 17
node', 11
node' (function), 7
nodes (function), 6, 12
nodes' (function), 13
noNodes (function), 6

out (function), 7
out' (function), 7
outdeg (function), 7
outdeg' (function), 7
outward graph Voronoi Diagram, 17

Path (type), 3, 16
path, 3
    root, 16
    shortest, 17
persistence, 19, 22
postorder, 12
postorder (function), 12
postorderF (function), 12
pre (function), 7
pre' (function), 7, 13
preorder, 12
preorder (function), 12
preorderF (function), 12

rdff (function), 14
rdff' (function), 14
rdfs (function), 13, 14
rdfs' (function), 14
reachable (function), 14
root path tree, 16
RootPath (module), 16
RoseTree (module), 12
RTree (type), 15, 16

scc (function), 14
shortest path, 15, 17
shortest path forest, 17
shortest path tree, 17
SP (module), 17
sp (function), 17
spanning tree
  breadth-first, 15
  depth-first, 11, 12
spLength (function), 17
spTree (function), 17
star (function), 6
strongly connected components, 12
suc (function), 7
suc' (function), 7, 10, 11, 13

topological sorting, 12
topsort (function), 14
Tree (data type), 12, 15
tree
  inward directed, 16
  labeled root path, 16
  minimum spanning, 18
  root path, 16
  shortest path, 17
  spanning
    breadth-first, 15
    depth-first, 11, 12
    minimum, 18

ucycle (function), 6
udff (function), 14
udff' (function), 14
udfs (function), 13, 14
udfs' (function), 14
UEdge (type), 3
ufold (function), 10, 12
undir (function), 10, 12, 19
undirected graph, 3, 18
unlab (function), 10, 12
unlabeled graph, 3
UNode (type), 3
UPath (type), 3

version tree, 22, 23
Voronoi Diagram, 17
Voronoi points, 17
Voronoi regions, 17
Voronoi set, 17

xdffWith (function), 14
xdfsWith (function), 14