Fully Persistent Graphs – Which One To Choose?

Martin Erwig

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

Abstract. Functional programs, by nature, operate on functional, or persistent, data structures. Therefore, persistent graphs are a prerequisite to express functional graph algorithms. In this paper we describe two implementations of persistent graphs and compare their running times on different graph problems. Both data structures essentially represent graphs as adjacency lists. The first uses the version tree implementation of functional arrays to make adjacency lists persistent. An array cache of the newest graph version together with a time stamping technique for speeding up deletions makes it asymptotically optimal for a class of graph algorithms that use graphs in a single-threaded way. The second approach uses balanced search trees to store adjacency lists. For both structures we also consider several variations, for example, ignoring edge labels or predecessor information.

1 Introduction

A data structure is called *persistent* if it is possible to access old versions after updates. It is called *partially persistent* if old versions can only be read, and it is called *fully persistent* if old versions can also be changed [4]. There is a growing interest in persistent data structures, for a recent overview, see [9]. However, persistent graphs have almost been ignored. In [6] we have sketched an implementation of unlabeled, fixed-size persistent graphs by functional arrays. The purpose of that paper was, however, to demonstrate an inductive view of graphs and a corresponding functional style of writing graph algorithms, mainly based on graph fold operations, and to show how this style facilitates reasoning about and optimization of graph algorithms.

In this paper we explain the implementation by functional arrays in more detail, and we extend it in several ways. First, we are now able to work with labeled graphs. Second, the implementation can also be used in semi-dynamic situations, that is, new nodes can be efficiently allocated, and graphs can grow to a limited degree. This has become possible through an extension of the graph representation by node partitions which are realized in a way similar to the implementation of sparse sets described in [3]. Third, we consider several specialized implementations: one for unlabeled graphs, one keeping only successor information, and one for a combination of both. Moreover, the underlying functional array implementation has been improved.

Besides the explanation of the persistent graph implementation based on functional arrays, the main goal of this paper is to find a good "standard representation" suitable for most application scenarios. We have therefore implemented persistent graphs also on the basis of balanced binary search trees. We present some example programs and report running times of the different graph implementations. We also pay attention to the question of whether specialized implementations (unlabeled, keeping only successors) are worthwhile. The results of this paper have helped us in the design of a *functional graph library*, which has been implemented in Standard ML and which can be downloaded from

```
http://www.fernuni-hagen.de/inf/pi4/erwig/fgl
```

All examples of this paper are contained in the distribution. The main contributions of this work are:

(1) The world's first functional graph library
(2) Several implementations of persistent graphs
(3) Empirical results about the performance of different persistent graph structures

Section 2 introduces a compact graph data type whose operations are briefly demonstrated with examples taken from graph reduction. These examples are used in Section 3 to explain the different implementations of persistent graphs. In Section 4 we describe a set of test programs and report running times of the different graph implementations. Conclusions follow in Section 5.

2 A Data Type for Graphs

In the following we consider directed node- and edge-labeled multi-graphs. This is a sufficiently general model of graphs, and many other graph types can be obtained as special cases: for instance, undirected graphs can be represented by symmetric directed graphs where "symmetric" means that presence of edge (v, w) implies the existence of edge (w, v). Unlabeled graphs have node and/or edge label type `unit`, and graphs embedded in the Euclidean plane can be modeled by having `real * real` node type.

Typical operations on graphs are the creation of an empty graph, adding, retrieving, and deleting nodes and edges, and retrieving and changing node and edge labels. We can cover all these functions by a simple interface consisting of just three operations. We have a type for nodes, which we assume for simplicity to be `int`, and a type for graphs whose type parameters `'a` and `'b` denote the type of node and edge labels, respectively.

```
type node = int
type ('a,'b) graph
```

Additionally, we use the following type abbreviations that make the typings of some operations more concise.

```
type      'b  adj     = ('b * node) list
type ('a,'b) context = 'b adj * node * 'a * 'b adj
```

Concerning operations, we have a constant[1] `empty` representing the empty graph, an operation `embed` that extends a graph by a new node together with incoming and outgoing edges, and an operation `match` that retrieves and at the same time removes a node with all its incident edges from the graph.

```
val empty : ('a,'b) graph
val embed : ('a,'b) context * ('a,'b) graph -> ('a,'b) graph
val match : node * ('a,'b) graph ->('a,'b)context*('a,'b)graph
```

Since graphs are not freely constructed by `empty` and `embed`, that is, since there are different term representations denoting the same graph, matching is, in general, not uniquely defined. The additional `node` parameter allows us to specify which representation is to be selected (namely that one with the given node inserted last) and thus makes `match` a function again; `match` is actually an example of an *active pattern* as described in [5]. If the node to be matched is not in the graph, a `Match` exception is raised.[2]

Consider, for example, the graph `g` in Figure 1 that represents the combinator expression (*sqr* 3) + (*sqr* 3). The node numbers are given for later reference.

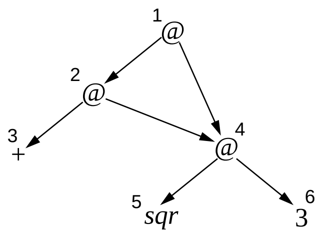

Fig. 1. Example Graph `g`

The type of `g` is `(combinator,direction) graph` with

```
datatype value = INT of int | BOOL of bool
datatype direction = L | R
datatype combinator =
   APP
 | COND
 | VAL of value
 | OP of (value * value -> value)
 | COMB of string
```

We have omitted the edge labels from the picture since the values are implied by the spatial embedding of the edges. The following expression constructs `g` in a bottom-up manner.

1. Note that in the array implementation `empty` takes an integer argument specifying the maximum graph size.
2. The reader might wonder why the argument node of `match` is also returned as a result (as part of the type `context`). It is for consistency with the operation `matchAny` (see Section 4.2) that does not take a node argument and therefore reports the node actually matched.

```
val g = foldr embed empty
        [([],1,APP,[(L,2),(R,4)]), ([],2,APP,[(L,3),(R,4)]),
         ([],3,OP plus,[]),        ([],4,APP,[(L,5),(R,6)]),
         ([],5,OP sqr,[]),         ([],6,VAL (INT 3),[])   ]
```

As already indicated this is indeed not the only way to build the graph g. Actually, we can insert the nodes in any order. For a precise definition of the semantics of graph constructors, see [6].

Suppose now we are to reduce g. We first have to reduce the subgraph rooted at node 4, and replace it with the result of the reduction. Thus, we first match node 4, and we get the context

```
(([(R,1),(R,2)],4,APP,[(L,5),(R,6)]),g')
```

where g' is any representation of the graph:

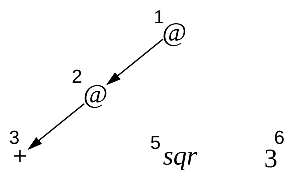

Fig. 2. Decomposed Graph g'

In this case we have to apply a δ-rule for computing sqr 3, and we re-insert node 4 with the result as the new node label, with the old predecessors, and with no successors, that is,

```
val r = embed (([(R,1),(R,2)],4,VAL (INT 9),[]),g’)
```

Thus we obtain the graph shown in Figure 3 in which nodes 5 and 6 have become garbage. Note that after the update g is still bound to the original graph; graph reduction as described here happens purely functionally, that is, node 4 is not destructively overwritten.

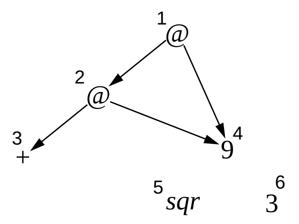

Fig. 3. Reduced Graph r

Summarizing we see that embed implements node and edge insertion and that match comprises the operations of finding and deleting nodes and edges. A match followed by embed can be used to realize update functions. It is not difficult to derive specific graph operations from these general ones. For instance, the insertion of a single labeled edge can be simply done by:

```
fun insEdge (v,w,l) g =
    let val ((p,_,vl,s),g') = match (v,g)
     in
        embed (p,v,vl,(l,w)::s,g')
    end
```

We will see the need for some more predefined graph operations when considering example programs in Section 4. Mostly on the grounds of efficiency we also provide the following functions.

```
val matchFwd : ('a,'b) graph -> 'a * 'b out
val matchAny : ('a,'b) graph -> ('a,'b) context * ('a,'b) graph
val isEmpty  : ('a,'b) graph -> bool
val newNodes : int -> ('a,'b) graph -> node list
```

The functions `matchFwd` and `matchAny` are just special versions of `match`: `matchFwd` selects only the node label and the successors, and `matchAny` matches an arbitrary node context. This means that `matchAny` is non-deterministic and can assume any valid term representation of its argument graph to return the outermost node context of this representation.The function `isEmpty` tests whether a graph is the empty graph, and `newNodes i g` generates a sequence of `i` nodes that are not contained in the graph `g`.

3 Persistent Graph Structures

The representation of a graph by (an array of) adjacency lists is often favored over the incindence matrix representation since its space requirement is linear in the graph size whereas an incidence matrix always needs $\Omega(n^2)$ space which is very wasteful for sparse graphs. Moreover, adjacency lists offer $O(k)$ access time to the k successors of an arbitrary node where we have to spend $\Omega(n)$ time to find the successors by scanning a complete row in an incidence matrix. We therefore focus on two alternatives for making adjacency lists persistent: the first representation uses a variant of the version tree implementation of functional arrays, and the second stores successor (and predecessor) lists in a balanced binary search tree. The functional array structure is more difficult to understand because it employs an additional cache array and a further array carrying a kind of time stamps for nodes. Moreover, to support some of the specialized operations efficiently, this structure is extended by a two-array implementation of node partitions to keep track of inserted and deleted nodes. We therefore explain this structure in some detail and only sketch the rather obvious binary tree implementation.

3.1 Implementation by Functional Arrays

In a first attempt we can make an imperative graph persistent by simply using a functional array for storing adjacency lists and node labels. However, with this representation the deletion of a node *v* from the graph (which is performed with every `match` operation) is quite complex, since we have to remove *v* from each of its predecessor's successor lists and from each of its successor's predecessor lists. This means, on the average, quadratic effort in the size of node contexts. To avoid this we therefore store with each node a positive integer when the node is present in the graph and a negative integer when the node is deleted. We also carry over positive node stamps into successor/predecessor lists. This has the following effect: When, for example, node *v* is deleted, we can simply set its stamp *s*(*v*) to -*s*(*v*); we need not remove *v* from all referencing successor and predecessor lists because when, say, a successor list *l* of *w* containing *v* is accessed, we can filter out all elements that have non-matching stamps, and by this *v* will not be returned as a successor. When *v* is re-inserted into the graph later, we set *s*(*v*) to |*s*(*v*)|+1, and 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* with stamp *s*(*v*) is still correctly ignored when *l* is accessed.

In the version tree implementation of functional arrays as described in [2] changes to the original array are recorded in an inward directed tree of (*index*, *value*) pairs that has the original array at its root. 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. By adding an imperative "cache" array to the leftmost node of the version tree the array represented by that node is actually duplicated. Since index access in 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 offering O(1) access all the time during the algorithm run.

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 via the version tree. Therefore, we create in our implementation a new cache for each new version derived from the original array.

An initial functional array representation of `g` is shown in Figure 4.

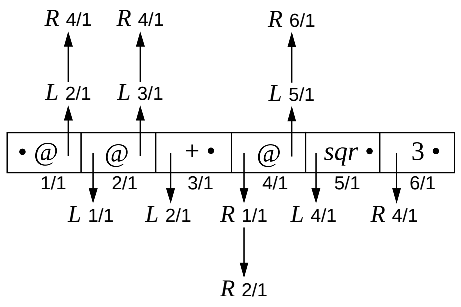

Fig. 4. Functional Array Representation of Graph `g`

Each array entry consists of a list of predecessors (extending downwards), a node label, and a list of successors (extending upwards). Empty lists are represented by •. Note that initially all node stamps are set to 1.

Now consider the result of the expression `match (4,g)`. First, the predecessor list, the node and its label, and the successor list are retrieved, and then the decomposed graph `g'` is constructed. Here, it suffices to add a single node to the version tree indicating that node 4 is deleted. If we had not the node stamps available, we would have to add four (!) more nodes to the version tree specifying the change of the successor lists (for nodes 1 and 2), respectively, predecessor lists (for nodes 5 and 6). The updated graph structure is shown in Figure 5.[3]

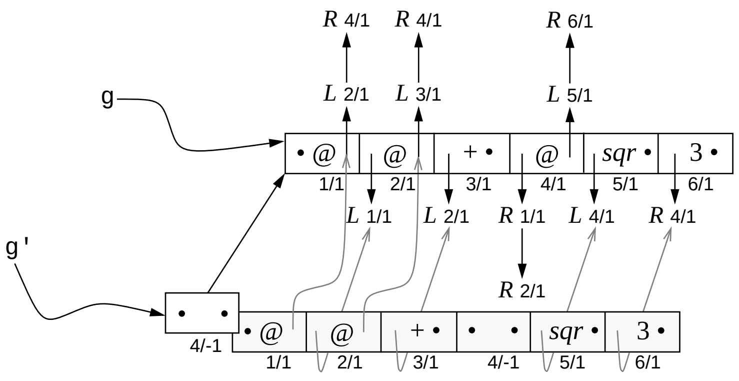

Fig. 5. Functional Array Representation of Graph `g'`

If we now access, say, the successors of node 2, we obtain just the list `[(L,3)]` – node 4 is omitted, since the node stamp found in the successor list does not match the stamp currently stored with the node.

Let us finally consider what happens if we re-insert node 4 to obtain the reduced graph `r` from Figure 3. First, the stamp of node 4 changes to +2, and a modification node with empty successor list and the old predecessors is added to the version tree. Unfortunately, we are not finished yet: we have to add node 4 to the successor lists of each of its predecessors, that is, we have to create corresponding nodes in the version tree. Note that in both these successor lists node 4 appears twice, but only the entry with the new stamp is relevant. The resulting graph representation is shown in Figure 6.

In principle, we had to update the predecessor lists for all successors, too, but this does not apply in this particular case, since the successor list is empty. The described updates actually use the array in a single-threaded way so that the version tree degenerates to a path with the cache at its end. If we now perform a further update to the original graph `g`, this is reflected in the version tree by a new sibling node of `g'`. Since this version was created from the original version `g`, the node is also equipped with its own

3. Note that the actual implementation differs in a minor point: instead of one array storing four-tuples we are working with four individual arrays for stamps, labels, predecessors, and successors. This simplifies the implementation at some points. On the other hand, the structure is easier to explain using only one array.

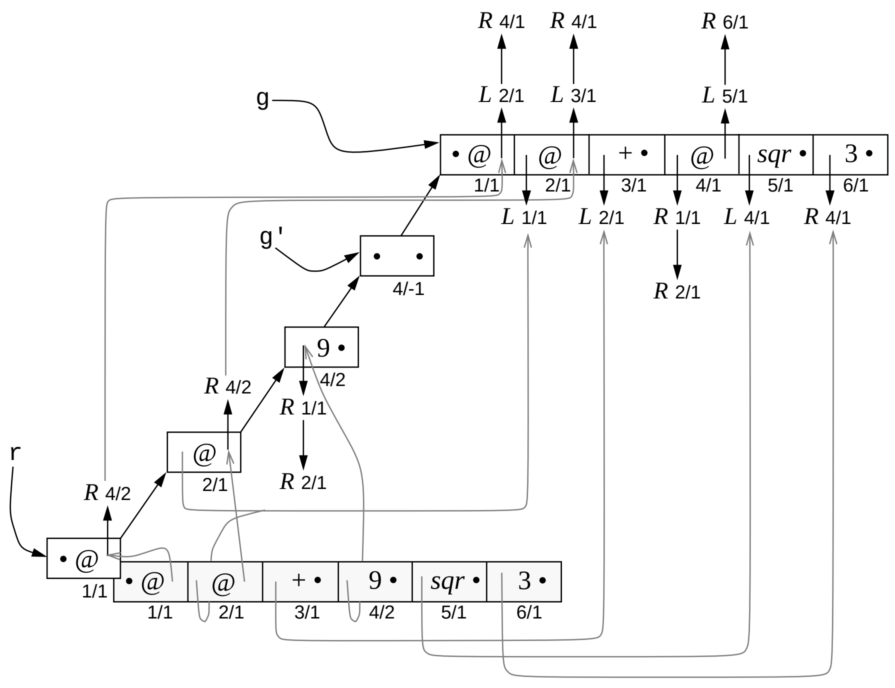

Fig. 6. Functional Array Representation of the Reduced Graph `r`

cache array to speed up single threaded operations on this version, too. This new cache is needed, since the other cache is still in use.

The array representation is expected to perform quite well in situations where graphs are used in a single-threaded way. However, in cases when the cache cannot be used the need for traversing possibly long paths in the version tree can slow down the structure significantly. Actually, few applications use graphs in a persistent manner. Maybe this is due to the fact that graph algorithms are traditionally expressed in an imperative style? In any case, a trivial, yet important, persistent use of graphs is given by executing more than one algorithm on a graph. By spending a new cache for each "primary" version, the functional array implementation is well prepared for these situations, too.

The functional array implementation is fine for the (minimal) interface described in Section 2. However, some algorithms require operations that cannot be easily reduced to the three basic ones and should therefore also be provided by a library. For instance, a simple algorithm for reversing the edges in a graph (see Section 4.2) uses the functions `matchAny` and `isEmpty`. Now in the array representation an implementation of `match-Any` is quite inefficient, since, in general, we have to scan the whole stamp array to find a valid, that is, non-deleted, node. The same is true for `isEmpty` and also for `newNodes`. Here the simple array implementation also requires, in general, linear time by scanning the whole array.

Thus to reasonably use the array implementation in those cases we have to extend the implementation to support these two operations in constant time.

We therefore 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*] ≤ *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], `isEmpty` is true if *k*=0, and `newNodes i` can simply return a list of nodes [*elem*[*k*+1], …, *elem*[*k*+`i`]]. Some other useful graph operations are efficiently supported by the node partition: *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 [3].

Although offering some nice features, the described extension has its drawbacks and limits: Keeping the partition information requires additional space and causes some overhead. Moreover, arrays do not become truly dynamic since they (at least in the current implementation) can neither grow nor shrink. In the following we will report test results for both the static array implementation (*Array*) and also for the semi-dynamic version (*DArray*) whenever this is possible.

The attentive reader 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. Although we have not studied the problem in detail, it seems that in practice, this is not 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.

3.2 Implementation by Balanced Binary Search Trees

We use the implementation of balanced binary search trees as provided by the SML/NJ library. Actually, this is the implementation described in [1]. For convenience, we have added two functions `update` and `findSome`. Although `update` can be realized by `remove` and `insert`, the directly implemented version saves multiple traversals of the same path.

Since a binary search tree can be used as a functional array implementation, we obtain an immediate realization of functional graphs, that is, a graph is represented by a pair (*t*, *n*) where *t* is a tree of pairs (*node*, (*predecessors*, *label*, *successors*)) and *n* is the largest node occurring in *t*. Note that *n* is used to support the operation `newNodes`. Keeping the largest node value used in the graph, this is possible in O(1) time.

3.3 Tuned Implementations

There are two principal sources for improving efficiency of any graph implementation: first, carrying edge labels in unlabeled graphs wastes unnecessarily space and causes some overhead. Second, keeping predecessor information is not necessary in algorithms that only access successors.

So in addition to the labeled/full-context array and tree implementations we have implemented unlabeled/full-context, labeled/forward, and unlabeled/forward versions to see which speed-ups can be achieved. It should be clear that the `match` operation is not available in the forward implementations. Instead an operation `matchFwd` giving only node label and successors is provided. Note that even `matchFwd` cannot be (efficiently) realized in a forward-tree implementation, since removing a node from a graph means to remove it from all referencing successor lists, but without the predecessor information, which provides direct access to all referencing nodes, all successor lists of the graph had to be scanned, which is not acceptable.

More extensions and improvements are conceivable, for example, combining the stamping technique with the tree implementation or making functional arrays fully dynamic. These will be examined in the near future.

4 Test Programs and Running Times

We have selected the following example programs to get a picture of the graph implementations' performances in different situations: depth-first search (`dfs`, Section 4.1), reversing the edges of a graph (`rev`, Section 4.2), generating cliques (`clique`, Section 4.3), combinator graph reduction (`red`, Section 4.4), and computing maximal independent sets of nodes (`indep`, Section 4.5). We cannot use the benchmarking kit Auburn [8] since (i) it is restricted to unary type constructors (labeled graphs have two type parameters), and (ii) argument and result types must not contain applications of other type constructors (`embed` and `match` take/yield *lists* of nodes (and labels)).

All programs have been run with SML of New Jersey 109.29 under Solaris 2 on a SUN SPARCstation 20 with 64MB main memory. Reported are the average user times (average taken over ten[4] runs). The benchmarks are intended to serve two goals: first, we would like to find out which graph implementation performs best in which scenario, and second, whether it is worthwhile to use specialized implementations in situations where only part of the functionality is needed.

With regard to the latter we consider three aspects: (1) *Kind of context*. This addresses the fact that there are graph algorithms (for example, *dfs*) that move only in forward direction through the graph. For all these, a graph representation that does not explicitly store predecessors is more space efficient and is expected to be faster because of less

4. Some implementations crashed due to memory shortage on large graphs. In those cases averages over three or five runs have been taken.

overhead. (2) *Edge Labels*. Many graph algorithms (for example, *dfs*, *rev*) do not need edge labels, that is, they also work on graphs without edge labels, and the question is whether they perform faster on a specialized graph type for unlabeled graphs. (3) *Graph Size*. For algorithms that do not change or at least do not increase the size of the graph the static array representation (without node partitions) is expected to run faster than semi-dynamic arrays.

For the comparison of the two main implementations we distinguish graph algorithms by the needed *operations on graphs*. Many algorithms use graphs in a “read-only” fashion, that is, they only decompose the graph by means of the `match` operation. On the other hand, there are algorithms that build or change graphs, and the performance picture might be different here. Algorithms for this case can be distinguished further according to whether they are *dynamic*, that is, generate new nodes and truly extend a graph, or *static*, that is, just perform operations on existing nodes and edges. Finally, although most algorithms seem to use graphs in a single-threaded way, there are examples of persistent graph uses, and it is interesting to know which structures are preferable in those situations.

The test programs provide examples for each kind of graph use, operation, and specialization:

Graph Use	*Operation*	*Specialization*	*Program*
single-threaded	`match`	*forward, unlab*	`dfs`
	`embed` (*static*)	*unlab*	`rev`
	`embed`(*dynamic*)	*unlab*	`clique`
			`red`
persistent	`embed`	*unlab*	`indep`

The test results provide some advice of the kind “If you have a graph problem of type *X*, then it is best to use graph implementation *Y*.”

4.1 Depth First Search

One of the most important general graph algorithm is certainly depth first search. We consider the task of computing a depth first spanning forest for a graph. We choose forests storing only nodes without labels; they are represented by:

```
datatype tree = BRANCH of node * tree list
```

Now a functional version of depth first search is defined by the two mutual recursive functions `dfs1` and `dfsn` which work as follows: First, `dfs1` decomposes `v` from `g` getting the list of successors (and node labels) `s` and the reduced graph `g1`. Then for each element of `s` (that is, for each second component) a depth first spanning tree is computed recursively by calling `dfsn`. Note that by using `g1` as the graph argument of `dfsn` it is ensured that `v` cannot be encountered again in a recursive call to `dfs1`. Finally, the constructor `BRANCH` is applied to `v` and to the resulting forest `f` yielding the spanning tree of `v`. In addition, the part of the graph not visited by the recursion is returned. Now `dfsn` does nothing more than calling `dfs1` for each node of its argument list, passing around decomposed graphs.

```
fun dfs1 (v,g) =
    let val ((_,_,_,s),g1) = match (v,g)
        val (f,g2) = dfsn (map (fn (_,w)=>w) s,g1)
     in
        (BRANCH (v,f),g2)
    end
and dfsn ([],g)   = ([],g)
 |  dfsn (v::l,g) =
    let val (t,g1) = dfs1 (v,g)
        val (f,g2) = dfsn (l,g1)
     in
        (t::f,g2)
    end
    handle Match => dfsn (l,g)
```

Since the graph might consist of unconnected components, the depth-first spanning forest is obtained by applying `dfsn` to all nodes of the graph:

```
fun dfs g = dfsn (nodes g,g)
```

The running times for `dfs` (user time in seconds, including garbage collection) are given in Table 1 for sparse graphs (of average degree 8) depending on the number of nodes in the graph and the chosen graph implementation.

	1 000	5 000	10 000	50 000	100 000	*Ratios*
Array	0.08	0.53	1.12	9.75	21.42	1
DArray	0.14	0.77	1.79	16.68	36.98	1.45 .. 1.75
Tree	0.25	1.63	3.62	29.61	67.01	3.04 .. 3.23

Table 1. Running Times for `dfs`

We see that both array implementations are always faster than the tree implementation and that *DArray* tends to take 70% more time than *Array*. It is interesting that *Tree* performs quite well with only a factor of about 3 slower than *Array*.

In Table 2 we show running times for unlabeled (*u*) and forward only (*f*) specialized implementations. We observe that omitting labels yields a speed-up of at least 7% (*Array*) or 12% (*Tree*) up to 20% (for both *Tree* and *Array*), and just keeping forward links (which is only possible with the array implementation) is about 3 to 18% faster. The unlabeled/forward implementation gives, as expected, the best performance and is 13 to 25% faster.

	1 000	5 000	10 000	50 000	100 000	*Ratios*
Array	0.10	0.53	1.12	9.75	21.42	1
Array^u	0.07	0.48	0.88	7.73	20.00	0.79 .. 0.93
Array^f	0.07	0.51	0.99	9.46	20.66	0.88 .. 0.97
Array^{uf}	0.06	0.41	0.84	7.53	18.70	0.75 .. 0.87
Tree	0.25	1.63	3.62	29.61	67.01	1
Tree^u	0.20	1.31	2.94	26.01	55.41	0.80 .. 0.88

Table 2. Speed-ups gained by Unlabeled and Forward Graphs

The improvements in running time obtained by the specialized implementations are similar for the other examples programs, so we shall not give any additional tables in the following.

4.2 Reversing Graphs

The second graph problem considered is to reverse the edges of a graph. The following simple algorithm can be used:

```
fun rev g =
    if isEmpty g then g else
    let val ((p,v,l,s),g') = matchAny g
     in
        embed ((s,v,l,p),rev g')
    end
```

We have already noticed that `isEmpty` and `matchAny` are efficiently supported only by the dynamic array implementation. Thus we omit the running times for the static version, since this would make `rev` actually a quadratic algorithm. The graphs used are the same as for the `dfs` tests.

	1 000	5 000	10 000	50 000	100 000	*Ratios*
DArray	0.27	2.12	4.35	47.27	120.59	1.04 .. 2.55
Tree	0.26	1.68	3.33	21.24	47.38	1

Table 3. Running Times for `rev`

It is striking that *Tree* outperforms the array implementation. This is certainly due to the fact that `matchAny` is realized by just taking the tree's root, which is very fast and, in particular, creates little garbage. It is therefore interesting to consider in this case the running times without garbage collection. Here the figure changes, and the tree implementation takes up to 50% more time, see Table 4.

	1 000	5 000	10 000	50 000	100 000	*Ratios*
DArray	0.23	1.18	2.37	12.88	24.32	1
Tree	0.24	1.42	2.88	16.79	36.36	1.04 .. 1.50

Table 4. Command Times for `rev` (without GC)

At this point it is important to mention that the test program for iterating the *DArray* implementation exhausted heap memory already on graphs with 50 000 nodes, and we were forced to take the average only over three runs. All in all this shows that worst-case running times given in big-Oh notation are one aspect and that actual running times as well as practicability and reliability of implementations are often a different matter.

Note that we can efficiently reverse graphs with the static array implementation if we rewrite the algorithm as follows.

```
fun revi ([],g)   = g
 |  revi (v::l,g) =
    let val ((p,_,lab,s),g') = match (v,g)
     in
        embed ((s,v,lab,p),revi (l,g'))
    end
    handle Match => g

fun revd g = revi (nodes g,g)
```

By avoiding the functions `matchAny` and `isEmpty` we obtain a linear algorithm. Deterministic matching causes *Tree* to spend more time on searching node contexts, and the static array implementation is again ahead, see Table 5.

	1 000	5 000	10 000	50 000	100 000	*Ratios*
Array	0.21	1.32	3.27	33.66	73.57	1
Tree	0.35	2.43	6.55	39.19	83.89	1.14 .. 2.00

Table 5. Running Times for `revd`

So `rev` is only of limited use in judging the implementations' efficiency for graph construction.

4.3 Generating Cliques

For a better comparison of the implementations of `embed`, we therefore use the following program[5] for generating cliques of a specified size (`map (fn x=>((),x))` just

5. For testing *DArray* we need a slightly different program taking into account that `empty` needs `n` as a size parameter.

extends the list of nodes by a unit edge label):

```
fun clique 0 = empty
 |  clique n =
    let val g   = clique (n-1)
        val l   = map (fn x=>((),x)) (nodes g)
        val [v] = newNodes 1 g
     in
        embed ((l,v,(),l),g)
    end
```

The results in Table 6 show that the tree implementation performs better in constructing graphs.

	50	100	500	1 000	*Ratios*
DArray	0.05	0.22	18.15	85.67	1.14 .. 2.00
Tree	0.03	0.17	8.45	46.69	1

Table 6. Running Times for `clique`

4.4 Combinator Graph Reduction

The graph type to be used for graph reduction was already given in Section 2. For lack of space we do not give here the complete code of the graph reducer.[6] We have implemented combinator graph reduction as described in Chapter 12 of [7]. The main components are the functions `unwind` and `red`: unwinding the graph's spine yields the combinator to be reduced together with all argument nodes and the root node to be replaced. The function `red` actually contains the reduction rules for all combinators. Consider the case for the *K* combinator: first, the argument nodes and the root node as given by `unwind` are bound to the variables `x`, `y`, and `r`. Then the root node `r` is matched in the graph `e` which is to be reduced. This is just to decompose `r` from `e` and to remember the references to it in the list `rp`. After that the label (`xl`) and the successors (`xs`) of `x` are determined by applying the function `fwd`. Note that `match` need not be used here, since `x` is not to be changed. Finally, node `r` is re-inserted, now with label and successors of `x`, but with its own predecessors (`rp`). This realizes, in a functional way, the "overwriting" of `r`. The situation is similar for the *S* combinator: bindings are created, the root is decomposed, and the new graph is built by node insertion. Note, however, that two new nodes (`n` and `m`) have to be generated. Performing this node generation *before* matching of `r` takes place ensures a single-threaded use of the graph, and we could, in principle, use an imperative graph implementation for the graph reducer. In contrast, if new nodes were generated in an imperative graph after matching `r`, `r` could be returned ("recycled") as one of the new nodes which would definitely lead to wrong results.

6. The code for all examples can be found in the FGL distribution.

```
fun red (v,e) =
    let val (comb,args) = unwind (v,[],e)
     in
        case comb of
          COMB "K" =>
            let val [x,y,r] = args
                val ((rp,_,_,_),e') = match (r,e)
                val (xl,xs) = fwd (x,e')
             in
                embed ((rp,r,xl,xs),e')
            end
        | COMB "S" =>
            let val [f,g,x,r] = args
                val [n,m] = newNodes 2 e
                val ((rp,_,_,_),e') = match (r,e)
             in
                embed ((rp,r,APP,[(L,n),(R,m)]),
                embed (([],n,APP,[(L,f),(R,x)]),
                embed (([],m,APP,[(L,g),(R,x)]),e')))
            end
        ...
    end
```

We give the running times for the reduction of graphs that represent applications of the fibonacci function on different arguments. We know that fibonacci has been criticized as a benchmark for graph reduction, but we are testing the underlying graph implementation, not the graph reducer, and for that fibonacci is sufficient by causing reductions and allocations. Unfortunately, *DArray* exhausts heap memory even for a single run when computing `fib 20`. For the smaller sizes, the tree implementation is clearly faster.

	`fib 10`	`fib 15`	`fib 20`	*Ratios*
DArray	0.52	11.91	—	1.33 .. 1.79
Tree	0.39	6.66	289.93	1
Reductions	1 031	11 576	128 521	
Allocations	814	9 139	101 464	

Table 7. Running Times for `red`

4.5 Maximal Independent Node Sets

The function `indep` for computing sets of non-adjacent nodes of maximal size is an example of an algorithm that uses graphs in a truly persistent way. Although the algorithm has exponential complexity (the problem is NP-hard) it is much faster than blindly trying all possible node subsets: in the second line a node `n` with maximum degree is determined. Then two node sets are computed, namely, the independent set of g simply without n, and n plus the independent set of g without n and all its neighbors. The larger of the two sets is the result.

```
fun indep g =
    if isEmpty g then [] else
    let val n = any (max,deg g) (nodes g)
        val ((p,_,_,l),g1) = match (n,g)
        val i1 = indep g1
        val i2 = n::indep (delNodes (l@p,g1))
     in
        if length i1>length i2 then i1 else i2
    end
```

Looking at the results in Table 8 it is striking that *Array* behaves very poorly on larger graphs. The reason is that `nodes` is linear in the size of the representation (that is, the size of the starting graph) and not linear in the number of nodes of the (usually much smaller) graph in the current recursion.

	15	20	50	*Ratios*
Array	0.02	4.00	2264.89	2.00 .. 8.68
DArray	0.02	1.85	555.34	1.97 .. 2.13
Tree	0.01	0.94	261.02	1

Table 8. Running Times for `indep`

5 Conclusions

The test results indicate that the tree implementation should be used as the primary representation of functional graphs. Although it is in some cases slower than the functional array implementation, it is more versatile and more reliable on large graphs. The tree implementation can also be easily translated into Haskell making the graph library accessible to a broader range of users, although the running times might be quite different due to the effects of lazy evaluation.

References

1. Adams, S.: Efficient Sets – A Balancing Act. *Journal of Functional Programming* **3** (1993) 553–561
2. Aasa, A., Holström, S., Nilsson, C.: An Efficiency Comparison of Some Representations of Purely Functional Arrays, *BIT* **28** (1988) 490–503
3. Briggs, P., Torczon, L.: An Efficient Representation for Sparse Sets, *ACM Letters on Programming Languages* **2** (1993) 59–69
4. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. *Journal of Computer and System Sciences* **38** (1989) 86–124

5. Erwig. M.: Active Patterns. *8th Int. Workshop on Implementation of Functional Languages*, LNCS 1268 (1996) 21–40
6. Erwig. M.: Functional Programming with Graphs. *2nd ACM SIGPLAN Int. Conf. on Functional Programming* (1997) 52–65
7. Field, A.J., Harrison, P.G.: *Functional Programming*. Addison-Wesley, Wokingham, England (1988)
8. Moss, G.E., Runciman, C.: Auburn: A Kit for Benchmarking Functional Data Structures. *9th Int. Workshop on Implementation of Functional Languages*, this LNCS volume (1997)
9. Okasaki, C.: Functional Data Structures, *Advanced Functional Programming*, LNCS 1129 (1996) 131–158