JFP **23** (2): 174–184, 2013.
doi:10.1017/S0956796812000457 First published online 7 January 2013

FUNCTIONAL PEARLS

An in-situ algorithm for expanding a graph

RICHARD S. BIRD

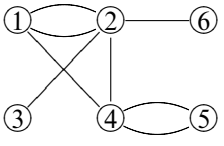
Department of Computer Science, Oxford University, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
(*e-mail:* bird@cs.ox.ac.uk)

1 Introduction

This pearl is devoted to a problem posed by Don Knuth about how to justify a certain array-based algorithm for changing the way an undirected graph is represented. In order to set the scene, we delay describing the precise problem until Section 4. Knuth (2011) recorded three different solutions, though no proofs of correctness were provided, nor even much explanation of why they worked. Recently, he asked various computer scientists interested in formal program development whether any of the proposed solutions "could have been discovered in a disciplined manner." In what follows we respond by developing a purely functional solution. The solution makes heavy use of the operations in the Haskell library *Data.Array*, and the whole exercise turns out to be a fascinating study of the arithmetic of graphs and arrays. One of the conditions of the problem is that the final algorithm has to be *in situ*, but we will get to that in due course.

2 Graphs and digraphs

Our objects of study are undirected graphs with no self-loops (edges from a vertex to itself), though multiple edges between two distinct vertices are allowed. In all that follows, suppose there are m edges and n vertices, labelled $[1 \,..\, n]$. For example, the following graph has $m = 8$ and $n = 6$:



One way to represent a graph is by an array that associates each vertex with a list of its adjacent vertices:

$$\begin{aligned} &\textbf{type } \mathit{Vertex} = \mathit{Int} \\ &\textbf{type } \mathit{Graph} = \mathit{Array}\ \mathit{Vertex}\ [\mathit{Vertex}] \end{aligned}$$

Exactly this representation is used in the Haskell library *Data.Graph*. For example, the graph above is represented by

$$g = \mathit{array}\ (1,6)\ [(1,[2,2,4]),(2,[1,1,3,4,6]),(3,[2]),(4,[1,2,5,5]),(5,[4,4]),(6,[2])]$$

The vertices and edges of a graph are defined by

$$\begin{aligned} &vertices\ g = indices\ g \\ &edges\ g \quad = [(j,k) \mid j \leftarrow vertices\ g,\ k \leftarrow g\,!j] \end{aligned}$$

The function *indices* returns the indices of an array in ascending order and (!) denotes array indexing. Each undirected edge connecting j and k occurs twice in *edges* g, once as (j,k) and once as (k,j). In particular, $length\ (edges\ g) = 2m$, not m.

Adjacency lists are assumed to be in ascending order. For g to be a valid graph it has to satisfy

$$g\,!i = [j \mid (j,k) \leftarrow edges\ g,\ k = i] \tag{1}$$

for all vertices i. For example, in the graph above, vertex 1 appears twice in the adjacency list for vertex 2 and once in the adjacency list for vertex 4, so its adjacency list is $[2,2,4]$. Of course, $g\,!i = [k \mid (j,k) \leftarrow edges\ g, j = i]$ by definition of *edges*.

A digraph is a graph with directed edges. There are 2^m ways of converting a graph into a digraph, two of which are to direct each edge from the smaller to the larger vertex, or to direct each edge from the larger to the smaller. The functions *directUp* and *directDn* of type $Graph \rightarrow Digraph$, where *Digraph* is a synonym for *Graph* though without the restriction (1), are defined by

$$\begin{aligned} &directUp\ g = array\ (bounds\ g)\ [(j, dropWhile\ (<j)\ (g\,!j)) \mid j \leftarrow vertices\ g] \\ &directDn\ g = array\ (bounds\ g)\ [(j, takeWhile\ (<j)\ (g\,!j)) \mid j \leftarrow vertices\ g] \end{aligned}$$

For our running example,

$$\begin{aligned} &directUp\ g = array\ (1,6)\ [(1,[2,2,4]),(2,[3,4,6]),(3,[\,]),(4,[5,5]),(5,[\,]),(6,[\,])] \\ &directDn\ g = array\ (1,6)\ [(1,[\,]),(2,[1,1]),(3,[2]),(4,[1,2]),(5,[4,4]),(6,[2])] \end{aligned}$$

Let us fix on *directUp* as the way of turning a graph into a digraph. Can we find an efficient method of going the other way? That is, can we define *mkGraph* to satisfy $mkGraph \cdot directUp = id$? Such a function is required as part of Knuth's problem.

We calculate for an arbitrary graph g:

$$\begin{aligned} & g\,!i \\ = \quad & \{\text{since } xs = takeWhile\ p\ xs +\!\!+ dropWhile\ p\ xs \text{ for any total } p\} \\ & takeWhile\ (<i)\ (g\,!i) +\!\!+ dropWhile\ (<i)\ (g\,!i) \\ = \quad & \{\text{definition of } directDn \text{ and } directUp\} \\ & (directDn\ g)!i +\!\!+ (directUp\ g)!i \\ = \quad & \{\text{specify } lower :: Digraph \rightarrow Digraph \text{ by } directDn = lower \cdot directUp\} \\ & (lower\ (directUp\ g))!i +\!\!+ (directUp\ g)!i \\ = \quad & \{\text{define } mkGraph \text{ by } (mkGraph\ d)!i = (lower\ d)!i +\!\!+ d\,!i\} \\ & (mkGraph\ (directUp\ g))!i \end{aligned}$$

Thus, we can define

$$mkGraph\ d = array\ (bounds\ d)\ [(i, (lower\ d)!i +\!\!+ d\,!i) \mid i \leftarrow vertices\ d] \tag{2}$$

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

provided we can find $lower :: Digraph \to Digraph$ to satisfy

$$(lower\ (directUp\ g))!i = takeWhile\ (<i)\ (g!i)$$

A suitable definition of *lower* is given by

$$(lower\ d)!i = [j \mid (j,k) \leftarrow edges\ d,\ k = i]$$

The right-hand side is identical to the right-hand side of (1), but remember that (1) does not hold when d is a digraph. To prove that *lower* meets its specification, we can argue:

$$\begin{aligned}
& (lower\ (directUp\ g))!i \\
= \quad & \{\text{definition of } lower\} \\
& [j \mid (j,k) \leftarrow edges\ (directUp\ g),\ k = i] \\
= \quad & \{\text{definition of } directUp \text{ and } edges\} \\
& [j \mid j \leftarrow vertices\ g,\ k \leftarrow dropWhile\ (<j)\ (g!j),\ k = i] \\
= \quad & \{\text{since } dropWhile\ (<j)\ (g!j) = filter\ (j<)\ (g!j);\ \text{see below}\} \\
& [j \mid j \leftarrow vertices\ g,\ k \leftarrow g!j,\ j < k \wedge k = i] \\
= \quad & \{\text{since } (j < k \wedge k = i) \equiv (j < i \wedge k = j)\} \\
& [j \mid j \leftarrow vertices\ g,\ k \leftarrow g!j,\ j < i \wedge k = i] \\
= \quad & \{\text{definition of } takeWhile\} \\
& takeWhile\ (<i)\ [j \mid (j,k) \leftarrow edges\ g,\ k = i] \\
= \quad & \{(1)\} \\
& takeWhile\ (<i)\ (g!i)
\end{aligned}$$

The equation $dropWhile\ (<j)\ (g!j) = filter\ (j<)\ (g!j)$ holds because the adjacency list $g!j$ is in ascending order and does not contain j. The absence of self-loops is critical to the success of the above calculation. Hence

$$\begin{aligned}
lower\ d = {} & array\ (bounds\ d)\ [(i, entry\ i) \mid i \leftarrow vertices\ d] \\
& \textbf{where } entry\ i = [j \mid (j,k) \leftarrow edges\ d,\ k = i]
\end{aligned}$$

Well and good, but definition (2) of *mkGraph* does not take linear time: *lower d* takes $\Theta(mn)$ steps rather than $\Theta(m+n)$ steps.

Fortunately, there is another function in *Data.Array* that comes to our aid. It is called *accum* and has type

$$accum :: Ix\ i \Rightarrow (e \to v \to e) \to Array\ i\ e \to [(i,v)] \to Array\ i\ e$$

The type constraint $Ix\ i$ restricts i to be an *index* type, such as *Int*, for naming the indices of the array. The first argument is an "accumulating" function for transforming array entries and values into new entries; the second argument is an array and the third argument is an association list of index-value pairs. The result is an array built by processing the association list from left to right, combining entries and values into new entries using the accumulating function. The process takes linear time in the length of the association list, assuming the accumulating function takes constant time.

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

That is what *accum* does in words. In symbols,

$$(accum\ f\ a\ jvs)!i = foldl\ f\ (a!i)\ [v \mid (j,v) \leftarrow jvs, j = i] \tag{3}$$

for all i in $range\ (bounds\ a)$. Well, not quite: there is an additional restriction on *jvs*, namely that every index j in this list should also lie in the range of a. If this condition does not hold, then the left-hand side returns an error while the right-hand side does not. I can appreciate why the Haskell library designers imposed the restriction, but it does spoil a good identity.

Now we can calculate for a digraph d:

$$\begin{aligned}
& (mkGraph\ d)!i \\
= \quad & \{\text{definition (2)}\} \\
& (lower\ d)!i +\!\!+ d!i \\
= \quad & \{\text{definition of } lower\} \\
& [j \mid (j,k) \leftarrow edges\ d,\ k = i] +\!\!+ d!i \\
= \quad & \{\text{since } foldr\ (:)\ ys\ xs = xs +\!\!+ ys\} \\
& foldr\ (:)\ (d!i)\ [j \mid (j,k) \leftarrow edges\ d,\ k = i] \\
= \quad & \{\text{since } foldr\ f\ e\ xs = foldl\ (flip\ f)\ e\ (reverse\ xs) \text{ for all finite } xs\} \\
& foldl\ (flip\ (:))\ (d!i)\ (reverse\ [j \mid (j,k) \leftarrow edges\ d,\ k = i]) \\
= \quad & \{\text{suppose } swap\ (j,k) = (k,j)\} \\
& foldl\ (flip\ (:))\ (d!i)\ (reverse\ [j \mid (k,j) \leftarrow map\ swap\ (edges\ d),\ k = i]) \\
= \quad & \{\text{distributing } reverse\} \\
& foldl\ (flip\ (:))\ (d!i)\ ([j \mid (k,j) \leftarrow reverse\ (map\ swap\ (edges\ d)),\ k = i]) \\
= \quad & \{(3)\} \\
& (accum\ (flip\ (:))\ d\ (reverse\ (map\ swap\ (edges\ d))))!i
\end{aligned}$$

Hence

$$mkGraph\ d = accum\ (flip\ (:))\ d\ (reverse\ (map\ swap\ (edges\ d))) \tag{4}$$

This pretty one-line definition of *mkGraph* does take linear time.

3 A second definition

Definition (4) is a testament to the expressive power of Haskell and the *Data.Array* library, but it is too highly structured for our purposes. Instead, we seek a more basic definition that takes the form of a simple loop involving array updates at a single point. The array update operation (//) in *Data.Array* has the general type

$$(//) :: Ix\ i \Rightarrow Array\ i\ e \rightarrow [(i,e)] \rightarrow Array\ i\ e$$

In particular, $a\ //\ [(i,e)]$ is an array identical to a except that it takes the value e at position i.

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

The function *accum* can be expressed in terms of (//). Assuming all indices in *ivs* are indices of *a*, we have

$$\begin{aligned} accum\ f\ a\ ivs &= foldl\ (update\ f)\ a\ ivs \\ update\ f\ a\ (i,v) &= a\ //\ [(i, f\ (a\,!i)\ v)] \end{aligned}$$

We now calculate an alternative definition of *mkGraph* in terms of *foldr* and (//). To do so, we make use of the following identities:

$$\begin{aligned} foldr\ f\ e \cdot concat &= foldr\ (flip\ (foldr\ f))\ e \\ foldr\ f\ e \cdot map\ g &= foldr\ (f \cdot g)\ e \\ foldr\ f\ e\ xs &= foldl\ (flip\ f)\ e\ (reverse\ xs) \end{aligned}$$

In the last identity *xs* is a finite list. Proofs can be found in Bird (1998). We calculate:

$$\begin{aligned} & mkGraph\ d \\ = \quad & \{(4)\} \\ & accum\ (flip\ (:))\ d\ (reverse\ (map\ swap\ (edges\ d))) \\ = \quad & \{\text{above definition of } accum\} \\ & foldl\ (update\ (flip\ (:)))\ d\ (reverse\ (map\ swap\ (edges\ d))) \\ = \quad & \{\text{since } foldr\ f\ e\ xs = foldl\ (flip\ f)\ e\ (reverse\ xs) \text{ for all finite } xs\} \\ & foldr\ (flip\ (update\ (flip\ (:))))\ d\ (map\ swap\ (edges\ d)) \\ = \quad & \{\text{since } foldr\ f\ e \cdot map\ g = foldr\ (f \cdot g)\ e\} \\ & foldr\ (flip\ (update\ (flip\ (:))) \cdot swap)\ d\ (edges\ d) \end{aligned}$$

Define $store = flip\ (update\ (flip\ (:))) \cdot swap$. Simplifying, we obtain

$$\begin{aligned} mkGraph\ d &= foldr\ store\ d\ (edges\ d) \\ store\ (j,k)\ g &= g\ //\ [(k,j\ : g\,!k)] \end{aligned}$$

We continue with the calculation:

$$\begin{aligned} & foldr\ store\ d\ (edges\ d) \\ = \quad & \{\text{definition of } edges\} \\ & foldr\ store\ d\ [(j,k) \mid j \leftarrow vertices\ d,\ k \leftarrow d\,!j] \\ = \quad & \{\text{list comprehensions}\} \\ & foldr\ store\ d\ (concat\ [[(j,k) \mid k \leftarrow d\,!j] \mid j \leftarrow vertices\ d]) \\ = \quad & \{\text{since } foldr\ f\ e \cdot concat = foldr\ (flip\ (foldr\ f))\ e\} \\ & foldr\ (flip\ (foldr\ store))\ d\ [[(j,k) \mid k \leftarrow d\,!j] \mid j \leftarrow vertices\ d] \\ = \quad & \{\text{define } arcs\ d\ j = [(j,k) \mid k \leftarrow d\,!j]\} \\ & foldr\ (flip\ (foldr\ store))\ d\ (map\ (arcs\ d)\ (vertices\ d)) \\ = \quad & \{\text{since } foldr\ f\ e \cdot map\ g = foldr\ (f \cdot g)\ e\} \\ & foldr\ (flip\ (foldr\ store) \cdot arcs\ d)\ d\ (vertices\ d) \end{aligned}$$

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

Define $step\ d = flip\ (foldr\ store) \cdot arcs\ d$. Simplifying, we obtain

$$\begin{aligned} mkGraph\ d &= foldr\ (step\ d)\ d\ (vertices\ d) \\ step\ d\ j\ g &= foldr\ (store\ j)\ g\ (d\,!j) \\ store\ j\ k\ g &= g\ //\ [(k, j\ :\ g\,!k)] \end{aligned}$$

Finally, we exploit the fact that $d = directUp\ g$ for some graph g, so $d\,!j$ is a list of vertices strictly greater than j. That means $d\,!j = g\,!j$, where

$$g = foldr\ (step\ d)\ d\ (dropWhile\ (\leqslant j)\ (vertices\ d))$$

It follows that we can drop the first argument of *step* and simplify *mkGraph* to read

$$mkGraph\ d = foldr\ step\ d\ (vertices\ d) \tag{5}$$

where $step\ j\ g = foldr\ (store\ j)\ g\ (g\,!j)$. As promised, definition (5) expresses *mkGraph* as a (nested) loop using single array updates.

4 Knuth's problem

Finally, we come to Knuth's problem. Our choice of representation for graphs is a jolly good one in a functional setting, but arguably less so in a language in which lists have to be implemented using linked structures and pointers. Another idea, dubbed the *sequential representation* by Knuth, is to represent elements of *Graph* and *Digraph* by two simpler arrays:

$$\textbf{type}\ SeqRep = (Array\ Int\ Vertex, Array\ Int\ Int)$$

Knuth called the sequential representation of $directUp\ g$ the *short code* of g. By the same token the full representation is called the *long code*. In either a long or short code (va, pa) of a graph with m edges and n vertices, the bounds of va are $(1, 2m)$ and the bounds of pa are $(0, n)$. Moreover, $pa\,!0$ is fixed to be 0. The values m and n can be determined from these bounds and, to save space, will be considered as constants in the functions that follow. The array va stores the concatenated list of adjacency lists and pa stores the boundaries that enable the concatenated adjacency lists to be reconstructed. More precisely, we can decode a sequential representation by

$$\begin{aligned} decode\ (va, pa) &= array\ (1, n)\ [(j, adj\ (va, pa)\ j)\ |\ j \leftarrow [1\ ..\ n]] \\ adj\ (va, pa)\ j &= map\ (va\,!)\ [pa\,!(j-1)+1\ ..\ pa\,!j] \end{aligned}$$

The inverse function *encode* is defined by setting $encode\ g = (va, pa)$, where

$$\begin{aligned} va &= array\ (1, 2{*}m)\ (zip\ [1\ ..\ 2{*}m]\ (concat\ (elems\ g))) \\ pa &= array\ (0, n)\ (zip\ [0\ ..\ n]\ (scanl\ (+)\ 0\ (map\ length\ (elems\ g)))) \end{aligned}$$

For our example graph g the sequential representation is a pair (va, pa) with elements

$$\begin{aligned} elems\ va &= [2, 2, 4, 1, 1, 3, 4, 6, 2, 1, 2, 5, 5, 4, 4, 2] \\ elems\ pa &= [0, 3, 8, 9, 13, 15, 16] \end{aligned}$$

while the sequential representation of $directUp\ g$ has elements

$$\begin{aligned} elems\ va &= [2, 2, 4, 3, 4, 6, 5, 5, -, -, -, -, -, -, -, -] \\ elems\ pa &= [0, 3, 6, 6, 8, 8, 8] \end{aligned}$$

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

The second half of *va* is ignored. Knuth's problem is to show how to reconstruct the long code from the short code, that is, to compute

$$expand = encode \cdot mkGraph \cdot decode$$

Furthermore, *expand* has to take linear time and be *in situ*, meaning that the total space available for computing *expand* is restricted to $2m + n + c$ units for some suitably small constant c. It requires $2m + n$ integers just to store the long code, so only a constant amount of extra space is allowed. That means arrays have to be modifiable in place and all recursive functions have to be translatable into stack-free loops. We solve the first problem by pretending that (//) is an in-place (and constant-time) operation, and the second by using only *foldl* and *foldr* to describe computations.

5 A solution

The core problem we have to tackle is how to store the concatenated elements of $mkGraph\, d$, where $d = decode\,(va, pa)$, in the array *va*. Suppose as an intermediate stage the elements of *d* have been installed in their correct final positions in *va*, while the remaining positions in *va* are filled with zeros. Furthermore, suppose *pa* has been adjusted so that

$$d\,!j = takeWhile\,(\neq 0)\,(map\,(va\,!)\,[pa\,!(j-1)+1\,..\,pa\,!j])$$

For our running example that means computing (va, pa) to satisfy

$$\begin{aligned} elems\ va &= [2,2,4,0,0,3,4,6,0,0,0,5,5,0,0,0] \\ elems\ pa &= [0,5,9,11,15,16,16] \end{aligned}$$

Call this process *installUp*. We then have to implement *installDn*, which puts the elements of *lower d* in their correct final places in *va* and readjusts *pa* to its final value.

To appreciate precisely how the elements of *pa* are changed, we need a little index arithmetic. Define $l(j), u(j)$ and $s(j)$ by

$$\begin{aligned} l(j) &= length\,((lower\,d)\,!j) \\ u(j) &= length\,(d\,!j) \\ s(j) &= sum\,[l(i) + u(i) \mid i \leftarrow [1\,..\,j]] \end{aligned}$$

The values $s(0), s(1), \ldots, s(n)$ are the final boundary positions to be stored in *pa*. Initially, $pa\,!j = u(1) + u(2) + \cdots + u(j)$ by definition of the short code. After *installUp* we have $pa\,!j = s(j) + l(j+1)$ for $1 \leqslant j < n$ and $pa\,!n = s(n)$ because the interval $[pa\,!(j-1)+1\,..\,pa\,!j]$ of *va* consists of the elements of $d\,!j$ followed by $l(j+1)$ zeros. That means that the elements of $(lower\,d)\,!k$ will go into positions $[pa\,!(k-1)-l(k)+1\,..\,pa\,!(k-1)]$. We can do this from right to left and at the same

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

time adjust pa so that it is in its final state by defining

$$\begin{aligned}
&installDn\ (va, pa) = foldr\ step\ (va, pa)\ [1\ ..\ n]\\
&step\ j\ (va, pa) \quad = foldr\ (store\ j)\ (va, pa)\ ks\\
&\quad \textbf{where}\ ks = takeWhile\ (\neq 0)\ (map\ (va\,!)\ [pa\,!(j-1)+1\ ..\ pa\,!j])\\
&store\ j\ k\ (va, pa)\ = (va\ //\ [(pa\,!(k-1), j)], pa\ //\ [(k-1, pa\,!(k-1)-1)])
\end{aligned}$$

This definition follows (5). Observe that $store\ j\ k$ both stores j in the slot $pa\,!(k-1)$ and decrements $pa\,!(k-1)$ ready for subsequent insertions. Thus, $pa\,!(k-1)$ is decremented $l(k)$ times, giving the correct final value of $pa\,!(k-1)$ for each k.

It remains to define *installUp*. We decompose this function into two components: (i) *installE*, which installs the edges of d in va; and (ii) *installD*, which completes the installation of d. Thus, $installUp = installD \cdot installE$.

The edges of $d = decode\ (va, pa)$ are given by

$$[(j, va\,!i)\ |\ j \leftarrow [1\ ..\ n],\ i \leftarrow [pa\,!(j-1)+1\ ..\ pa\,!j]]$$

The two vertices of each edge are stored in adjacent slots in va. The code for *installE* is straightforward and we would not derive it:

$$\begin{aligned}
&installE\ (va, pa) = foldr\ step\ (va, pa)\ [1\ ..\ n]\\
&step\ j\ (va, pa) \quad = foldr\ store\ (va, pa)\ [pa\,!(j-1)+1\ ..\ pa\,!j]\\
&\quad \textbf{where}\ store\ i\ (va, pa) = (va\ //\ [(2{*}i-1, j), (2{*}i, va\,!i)], pa)
\end{aligned}$$

The array pa is carried along but is unchanged. The main point of interest about *installE* is that edges are installed in the array va from right to left. In a short code, the second half of va is unused and therefore available for updates without destroying essential information stored in the lower half. The right-to-left computation using *foldr* guarantees that $va\,!i$ refers to the original entry in va, not to any updated element. The guarantee holds because $i \leqslant 2i-1$ for $1 \leqslant i$.

That leaves *installD*. As a result of *installE*, we have

$$\begin{aligned}
&(lower\ d)\,!j = [va\,!(2{*}i-1)\ |\ i \leftarrow [1\ ..\ m],\ va\,!(2{*}i) = j]\\
&d\,!j \qquad\quad = [va\,!(2{*}i)\ |\ i \leftarrow [1\ ..\ m],\ va\,!(2{*}i-1) = j]
\end{aligned}$$

In fact, the elements of va in odd positions will be in ascending order, so

$$d\,!j = [va\,!(2{*}i)\ |\ i \leftarrow takeWhile\ (\lambda i \rightarrow va\,!(2{*}i-1) = j)\ [p+1\ ..\ m] \tag{6}$$

where p is the largest index of va such that $va\,!(2{*}p-1) < j$.

The elements of $d\,!j$ have to go into va at positions $[s(j-1)+l(j)+1..s(j)]$, followed by $l(j+1)$ zeros in positions $[s(j)+1\ ..\ s(1)+l(j+1)]$ for each j. To do so, we need to know $l(j)$. The following calculation makes use of a variant of *accum* called *accumArray* and which is defined by

$$accumArray\ f\ e\ bnds\ ivs = accum\ f\ (array\ bnds\ [(i, e)\ |\ i \leftarrow range\ bnds])\ ivs$$

This function is also provided in the library *Data.Array*. We calculate:

$$\begin{aligned}
&l(j)\\
=\ &\{\text{definition}\}\\
&length\ [va\,!(2{*}i-1)\ |\ i \leftarrow [1\ ..\ m],\ va\,!(2{*}i) = j]
\end{aligned}$$

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{\text{since } length = foldl\,(+)\,0 \cdot map\,(const\ 1)\} \\
 & foldl\,(+)\,0\,[1 \mid i \leftarrow [1\,..\,m],\ va\,!(2{*}i) = j] \\
= & \{(3) \text{ and definition of } accumArray\} \\
 & (accumArray\,(+)\,0\,(0,n)\,[(va\,!(2{*}i),1)\ i \leftarrow [1\,..\,m]])\,!j
\end{array}
$$

Hence, $l(j) = la\,!j$, where

$$la = accumArray\,(+)\,0\,(0,n)\,[(va\,!(2{*}i),1)\ i \leftarrow [1\,..\,m]]$$

The array *la* can occupy the same space as *pa* because *pa* is no longer needed once the edges are installed. The definition of *la* can be translated into one that uses single array updates, but we omit details.

To show how to define *installD* we first show how to compute a digraph from its edges. We reason:

$$
\begin{array}{ll}
 & d\,!i \\
= & \{\text{definition in terms of } edges\} \\
 & [k \mid (j,k) \leftarrow edges\,d, j = i] \\
= & \{\text{since } foldl\ snoc\ [\,]\ xs = xs \text{ for finite } xs, \text{ where } snoc\ xs\ x = xs +\!\!+ [x]\} \\
 & foldl\ snoc\ [\,]\ [k \mid (j,k) \leftarrow edges\,d, j = i] \\
= & \{(3), \text{setting } empty\,d = array\,(bounds\,d)\,[(i,[\,]) \mid i \leftarrow vertices\,d]\} \\
 & (accum\ snoc\ (empty\,d)\,(edges\,d))\,!i
\end{array}
$$

Using the definition of *accum* in terms of (//) given in Section 3, we arrive at

$$
\begin{array}{ll}
d & = foldl\ update\ (empty\,d)\,(edges\,d) \\
empty\,d & = array\,(bounds\,d)\,[(i,[\,]) \mid i \leftarrow vertices\,d] \\
update\,d\,(j,k) & = d\ //\ [(j, d\,!j +\!\!+ [k])]
\end{array}
$$

We continue the calculation:

$$
\begin{array}{ll}
 & foldl\ update\ (empty\,d)\,(edges\,d) \\
= & \{\text{definition of } edges\} \\
 & foldl\ update\ (empty\,d)\,[(j,k) \mid j \leftarrow vertices\,d,\ k \leftarrow d\,!j] \\
= & \{\text{list comprehensions}\} \\
 & foldl\ update\ (empty\,d)\,(concat\ [[(j,k) \mid k \leftarrow d\,!j] \mid j \leftarrow vertices\,d]) \\
= & \{\text{since } foldl\,f\,e \cdot concat = foldl\,(foldl\,f)\,e\} \\
 & foldl\,(foldl\ update)\,(empty\,d)\,[[(j,k) \mid k \leftarrow d\,!j] \mid j \leftarrow vertices\,d] \\
= & \{\text{with } arcs\,d\,j = [(j,k) \mid k \leftarrow d\,!j]\} \\
 & foldl\,(foldl\ update)\,(empty\,d)\,(map\,(arcs\,d)\,(vertices\,d)) \\
= & \{\text{since } foldl\,f\,e \cdot map\,g = foldr\,(\lambda y\ x \rightarrow f\ y\,(g\ x))\,e\} \\
 & foldl\,(\lambda g\ j \rightarrow foldl\ update\ g\,(arcs\,d\,j))\,(empty\,d)\,(vertices\,d)
\end{array}
$$

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

Hence, defining $step\ d\ g\ j = foldl\ update\ g\ (arcs\ d\ j)$ and simplifying, we arrive at

$$\begin{array}{ll} d & = foldl\ (step\ d)\ (empty\ d)\ (vertices\ d) \\ step\ d\ g\ j & = foldl\ update\ g\ [(j,k) \mid k \leftarrow d\,!j] \\ empty\ d & = array\ (bounds\ d)\ [(i,[\,]) \mid i \leftarrow vertices\ d] \\ update\ d\ (j,k) & = d\ //\ [(j, d\,!j +\!\!+ [k])] \end{array}$$

The definition of *installD* follows this scheme, except that it also maintains the value p as described in (6):

$$\begin{array}{l} installD\ (va,pa) \quad = fst\ (foldl\ step\ ((va,la),0)\ [1\,..\,n]) \\ \quad \textbf{where}\ la = accumArray\ (+)\ 0\ (0,n)\ [(va\,!(2{*}i),1)\ i \leftarrow [1\,..\,m]] \\ step\ ((va,pa),p)\ j \quad = foldl\ (store\ j)\ ((va, pa\ //\ [(j, pa\,!(j{-}1))]), p{+}u)\ (vs +\!\!+ zs) \\ \quad \textbf{where}\ vs = [va\,!(2{*}i) \mid i \leftarrow takeWhile\ (\lambda i \rightarrow va\,!(2{*}i{-}1) = j)\ [p{+}1\,..\,m]] \\ \qquad\quad zs = replicate\ (\textbf{if}\ j < n\ \textbf{then}\ pa\,!(j{+}1)\ \textbf{else}\ 0)\ 0 \\ \qquad\quad u\ \ = length\ vs \\ store\ j\ ((va,pa),p)\ v = ((va\ //\ [(pa\,!j{+}1, v)], pa\ //\ [(j, pa\,!j{+}1)]), p) \end{array}$$

At the beginning of $step\ ((va,pa),p)\ j$, we have $pa\,!i = s(i{-}1) + l(i)$ for $i < j$ and $pa\,!i = l(i)$ for $i \geqslant j$. At the conclusion we have

$$pa\,!j = s(j{-}1) + l(j) + u(j) + l(j{+}1) = s(j) + l(j{+}1)$$

The critical observation is that, since $s(j{-}1)$ is the total length of the full adjacency lists for vertices less than j, and $l(j)$ is the number of vertices less than j adjacent to j, and $2p$ is the number of edges involving a vertex less than j, we have the invariant $s(j{-}1) + l(j) \leqslant 2p$ at the beginning of $step\ ((va,pa),p)$. Hence (6) remains true after installing $g\,!1, g\,!2, \ldots g\,!(j{-}1)$.

Putting all the pieces together, we now have

$$expand = installDn \cdot installUp$$

This is our solution to Knuth's problem.

6 Concluding remarks

Knuth posed the short-to-long problem as an exercise in the *Journal of Algorithms* in 1990. The solution he originally thought of was "somewhat tricky." A second solution was submitted by Mihaela Juganaru, but never published as the problems section of the *Journal of Algorithms* became dormant in 1992. Knuth later found a simpler solution, though it was twice as slow as the first. Our final algorithm resembles Knuth's second solution, though it is different in a number of respects.

As to whether the algorithm "could have been discovered in a disciplined manner," that is really for the reader to judge. We have been at pains to make the development as calculational as possible. The functions *accum* and *accumArray* featured in many of the more interesting calculations. The latter operation made an appearance in the very first pearl of my book (Bird, 2010), as well as in two subsequent pearls. It is rapidly becoming my favourite function. The real interest in Knuth's problem is not the solution, which is not particularly attractive, but the calculational properties of the functions in *Data.Array* that led to it.

https://doi.org/10.1017/S0956796812000457 Published online by Cambridge University Press

Finally, I would like to thank Jeremy Gibbons and two referees for their patience in dealing with the different drafts of this paper. Their positive suggestions and other feedback greatly helped in preparing the final version.

References

Bird, R. S. (1998) *Introduction to Functional Programming using Haskell*. London: Prentice Hall.

Bird, R. S. (2010) *Pearls of Functional Algorithm Design*. Cambridge, UK: Cambridge University Press.

Knuth, D. E. (2011) Solutions to a puzzling problem. Extract from *A Companion to the Papers of Donald Knuth*. Stanford, CA: California Centre for the Study of Languages and Information. Available at: http://www-cs-faculty.stanford.edu/ knuth/shortcode.pdf