# FUNCTIONAL PEARL

## *The unweighted average shortest path*

Matthew Naylor[1], Simon Moore[1], Andrey Mokhov[2]
Univesity of Cambridge[1], University of Newcastle[2]

### 1 Overview

We consider the problem of finding the average shortest path in an unweighted graph. That is, the average path length in the set of all shortest paths starting at each vertex and ending at each other vertex. This is an important metric in the biomedical industry where it is used to measure the robustness of protein interaction networks (Mokhov, 2017). We start out with a simple, seemingly naive, solution based on iterative deepening and then calculate a more efficient version that exploits memoisation. On highly-connected graphs provided by our industrial partners, this algorithm turns out to be significantly faster than a standard solution based on repeated breadth-first search.

### 2 Data types

We represent graphs by immutable arrays that map each integer vertex to a list of neighbouring vertices.

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

We write $g\ !\ v$ to obtain the neighbours of vertex $v$ in graph $g$, and $\mathit{indices}\ g$ to obtain a list of all vertices in $g$. For simplicty, we assume that vertices lie in the integer range $0 \ldots (n-1)$ where $n = \mathit{length}\ (\mathit{indices}\ g)$.

We also require a *Set* data type, for storing sets of vertices, with an efficient union operation. In our experiments, we have found that a simple bit-vector representation, shown in Figure 1, works very well. Although the time complexity of bit-vector union (bitwise-or) is linear in the maximum set size (that is, even when the sets are sparse), this is offset by bit-level parallalism and excellent spatial locality. And as we shall see, vertex sets tend to grow rapidly towards the maximum size when processing highly-connected graphs.

### 3 Iterative deepening

Let us first consider the problem of finding all vertices that are reachable from a given vertex $v$ in $d$ hops or less. In classic functional programming style, we can define this function by structural recursion on $d$. The set of vertices reachable from $v$ in 0 hops is simply the singleton set $\{v\}$. And the set of vertices reachable from $v$ in $d+1$ hops or less

```
type Set      =  Integer

singleton     ::  Vertex → Set
singleton v   =  1 `shiftL` v

unions        ::  [Set] → Set
unions        =  foldr (.|.) 0

size          ::  Set → Int
size          =  popCount
```

Fig. 1. A vertex set implementation using arbitrary precision integers and GHC's *Data.Bits* library.

is the union of the sets that are reachable from each neighbour of $v$ in $d$ hops or less, plus $v$ itself. We call this function *dfs* as it implements *bounded depth-first search*.

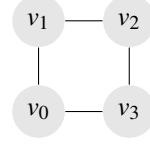
```
dfs             ::  Graph → Int → Vertex → Set
dfs g 0 v       =  singleton v
dfs g (d + 1) v =  combine g (dfs g d) v

combine         ::  Graph → (Vertex → Set) → Vertex → Set
combine g f v   =  unions (singleton v : map f (g ! v))
```

To obtain the vertices reachable at all distances, we can repeatedly call *dfs* with an increasingly larger bound. This is classic *iterative deepening*.

```
itd       ::  Graph → Vertex → [Set]
itd g v   =  [dfs g d v | d ← [0..]]
```

When two consecutive sets in the result list of *itd* are equal, then a fixed point has been reached and all reachable vertices have been discovered. To illustrate, if we apply *itd* to vertex $v_0$ in the graph

*box* =




then we obtain the following infinite list of vertex sets.

$$[\{v_0\}, \{v_0, v_1, v_3\}, \{v_0, v_1, v_2, v_3\}, \{v_0, v_1, v_2, v_3\}, \cdots]$$

At $d = 3$, a fixed point is observable. Now, taking the differences between consecutive sets in this list, we get

$$[\{v_1, v_3\}, \{v_2\}, \{\}, \{\}, \cdots]$$

This contains the lengths of all shortest paths starting from $v_0$ in order of increasing length: the shortest paths to $v_1$ and $v_3$ are of length 1, and the shortest path to $v_2$ is of length 2.

Iterative deepening is certainly a simple way to compute shortest paths – perhaps even the simplest possible way – but it might be considered inefficient: the search begins from sratch every time $d$ is incremented, and no effort is made to avoid visiting the same vertex many times. Standard breadth-first search would overcome these problems, but for now we persist with iterative deepening.

## 4 Memoisation

Given that we need to compute the shortest paths for *all* (source, destination) vertex pairs, a tempting next step is to define a function which, given a graph $g$, applies iterative deepening at each vertex in turn:

$$[itd\ g\ v \mid v \leftarrow indices\ g]$$

What we have here, if we unfold the definition of *itd*, is a nested loop: for each vertex $v$, for each distance $d$, compute the vertices reachable from $v$ in $d$ hops or less. We can compute the same information, albeit in a different order, if we transpose the loops.

$$\begin{array}{lcl} level & :: & Graph \rightarrow Int \rightarrow [Set] \\ level\ g\ d & = & [dfs\ g\ d\ v \mid v \leftarrow indices\ g] \end{array}$$

$$\begin{array}{lcl} levels & :: & Graph \rightarrow [[Set]] \\ levels\ g & = & [level\ g\ d \mid d \leftarrow [0..]] \end{array}$$

With this formulation, it becomes clear that we can compute $level\ g\ d$ in full before we compute $level\ g\ (d+1)$. This is useful because $level\ g\ (d+1)$ depends on $dfs\ g\ (d+1)\ v$, for each vertex $v$, which in turn depends on $dfs\ g\ d\ w$, for each vertex $w$ that is a neighbour of $v$, and these sets have already been computed by $level\ g\ d$. To exploit this, we would like a function $step\ g\ l$ such that:

$$step\ g\ (level\ g\ d) \ = \ level\ g\ (d+1) \tag{1}$$

That is, a function to compute $level\ g\ (d+1)$ given $level\ g\ d$. We derive such a function by basic program calculation. The only law we require to do so is:

$$level\ g\ d\ !!\ v \ = \ dfs\ g\ d\ v \tag{2}$$

where $l$ !! $i$ returns the $i^{th}$ element of the list $l$. This law is true under our assumption that $indices\ g\ =\ [0..(n-1)]$ where $n$ is the number of vertices. The calculation proceeds as follows.

$$
\begin{array}{ll}
 & step\ g\ l \\
= & \quad \{\ \text{Equation (1), assuming } l\ =\ level\ g\ d\ \} \\
 & level\ g\ (d+1) \\
= & \quad \{\ \text{Definition of } level\ \} \\
 & [dfs\ g\ (d+1)\ v \mid v \leftarrow indices\ g] \\
= & \quad \{\ \text{Definition of } dfs\ \} \\
 & [combine\ g\ (dfs\ g\ d)\ v \mid v \leftarrow indices\ g] \\
= & \quad \{\ \text{Equation (2)}\ \} \\
 & [combine\ g\ (level\ g\ d\ !!)\ v \mid v \leftarrow indices\ g] \\
= & \quad \{\ \text{Since } l\ =\ level\ g\ d\ \} \\
 & [combine\ g\ (l\ !!)\ v \mid v \leftarrow indices\ g]
\end{array}
$$

We end up with the following definition.

$$
\begin{array}{lcl}
step & :: & Graph \to [Set] \to [Set] \\
step\ g\ l & = & [combine\ g\ (l\ !!)\ v \mid v \leftarrow indices\ g]
\end{array}
$$

We would also like a function *initial* $g$, equal to *level* $g$ 0, which after beta-reduction can be defined as:

$$
\begin{array}{lcl}
initial & :: & Graph \to [Set] \\
initial\ g & = & [singleton\ v \mid v \leftarrow indices\ g]
\end{array}
$$

Now we can redfine the *levels* function, this time exploiting memoisation, as

$$
\begin{array}{lcl}
levels & :: & Graph \to [[Set]] \\
levels\ g & = & iterate\ (step\ g)\ (initial\ g)
\end{array}
$$

As a final efficiency improvement, we can replace the $O(n)$ list index function (!!) with the $O(1)$ array index function (!), as shown in Figure 2.

## 5 The average shortest path

Let us return to our earlier example, applying iterative deepening at vertex $v_0$ in the *box* graph to get:

$$sets\ =\ [\{v_0\}, \{v_0, v_1, v_3\}, \{v_0, v_1, v_2, v_3\}, \{v_0, v_1, v_2, v_3\}, \cdots]$$

The lengths of the shortest paths from $v_0$ can be computed by taking the difference in size between consecutive elements in this list, stopping when the difference becomes zero.

```
toArray     ::  [Set] → Array Vertex Set
toArray xs  =   listArray (0, length xs − 1) xs

step        ::  Graph → Array Vertex Set → Array Vertex Set
step g l    =   toArray [combine g (l !) v | v ← indices g]

initial     ::  Graph → Array Vertex Set
initial g   =   toArray [singleton v | v ← indices g]

levels      ::  Graph → [[Set]]
levels g    =   map elems (iterate (step g) (initial g))
```

Fig. 2. Final version of the *levels* function, using $O(1)$ array indexing instead of $O(n)$ list indexing.

```
diffs     ::  [Int] → [Int]
diffs xs  =   takeWhile (≠ 0) (zipWith (−) (tail xs) xs)
```

For example, *diffs* (*map size sets*) yields $[2,1]$. That is, two shortest paths of length one, and one of length two. To compute the sum of all of the shortest path lengths found by iterative deepening, we define

```
total   ::  [Set] → Int
total   =   sum · zipWith (∗) [1..] · diffs · map size
```

Recall that we have arrived at our definition of *levels g* by transposing the matrix obtained by applying iterative deepening at each vertex. So we can now define the sum of all shortest paths in a graph as

```
ssp   ::  Graph → Int
ssp   =   sum · map total · transpose · levels
```

Note that we have to define our own *transpose* function as the one provided by GHC is not sufficiently lazy. Finally, we can define the average shortest path of a graph as

```
asp     ::  Graph → Double
asp g   =   let n = length (indices g) in ssp g // (n ∗ (n − 1))
```

where $x \mathbin{/\!/} y$ divides integer $x$ by integer $y$ to yield a double-precision floating-point result. This definition assumes that $g$ is connected – every node is reachable from every other node.

## 6 Breadth-first search

Breadth-first search (BFS) is a textbook solution to find the shortest paths in an unweighted graph and therefore makes an interesting point of comparison. One way to implement BFS is to maintain a *visited set* and a *fringe*. The visited set contains all vertices that have been visited so far, and the fringe is a list of visited vertices whose neighbours have not yet been visited.

$$\textbf{type}\ \mathit{BFSState} \quad = \quad (\mathit{IntSet},\ [\mathit{Vertex}])$$

We represent the visited set using Haskell's *IntSet*, which has efficient insertion and membership functions for sets containing integers. When we visit one of the neighbours of a fringe vertex, we update the visited set and fringe list as follows:

$$\begin{array}{lcl}
\mathit{visit} & :: & \mathit{BFSState} \to \mathit{Vertex} \to \mathit{BFSState} \\
\mathit{visit}\ (\mathit{visited}, \mathit{fringe})\ v & & \\
\quad \mid \mathit{member}\ v\ \mathit{visited} & = & (\mathit{visited}, \mathit{fringe}) \\
\quad \mid \textbf{otherwise} & = & (\mathit{insert}\ v\ \mathit{visited},\ v : \mathit{fringe})
\end{array}$$

In each step of the BFS, we compute a new fringe based on the current one by visiting the neighbours of all fringe vertices.

$$\begin{array}{lcl}
\mathit{step} & :: & \mathit{Graph} \to \mathit{BFSState} \to \mathit{BFSState} \\
\mathit{step}\ g\ (\mathit{visited}, \mathit{fringe}) & = & \mathit{foldl}\ \mathit{visit}\ (\mathit{visited}, [])\ (\mathit{concatMap}\ (g\ !)\ \mathit{fringe})
\end{array}$$

We can now define a *bfs* function that computes the fringe list at each step.

$$\begin{array}{lcl}
\mathit{bfs} & :: & \mathit{Graph} \to \mathit{Vertex} \to [[\mathit{Vertex}]] \\
\mathit{bfs}\ g\ v & = & \mathit{map}\ \mathit{snd}\ (\mathit{iterate}\ (\mathit{step}\ g)\ (\mathit{singleton}\ v,\ [v]))
\end{array}$$

By applying this function at each vertex in turn, we can compute all shortest paths and hence the average shortest path. We refer to this as *repeated breadth-first search*.

## 7 Time complexity

The classic time complexity of BFS is $O(|V| + |E|)$, since every vertex is visited and every edge traversed. However, this assumes $O(1)$ time *insert* and *member* functions. Using *IntSet*, the time complexity of *insert* and *member* is actually bounded by $|W|$, the number of bits in an *Int* (Okasaki and Gill, 1998). This means the overall time complexity is $O(|V| \times |W| + |E| \times |W|)$, since *insert* is called for each vertex, and *member* is called for each edge. A possible alternative would be to use a set representation that allows $O(log(|V|))$ time *insert* and *member* functions, but in our experience *IntSet* performs better in practice.

Table 1. *Performance of breadth-first search and memoised iterative deepening.*

| **Graph** | **g0** | **g1** | **g2** | **g3** | **g4** |
|---|---|---|---|---|---|
| **Vertices** | 1628 | 3487 | 4000 | 7000 | 10000 |
| **Edges** | 53406 | 115898 | 80000 | 140000 | 200000 |
| **Diameter** | 5 | 5 | 4 | 5 | 5 |
| **Haskell BFS Runtime (s)** | 2.544 | 15.836 | 15.100 | 68.600 | 195.656 |
| **Haskell MID Runtime (s)** | 0.028 | 0.088 | 0.092 | 0.272 | 0.440 |
| **Speedup Factor** | 91 | 180 | 164 | 252 | 445 |
| **C BFS Runtime (s)** | 0.120 | 0.583 | 0.625 | 2.090 | 4.521 |
| **C MID Runtime (s)** | 0.005 | 0.024 | 0.020 | 0.101 | 0.185 |
| **Speedup Factor** | 24 | 24 | 31 | 21 | 24 |

Repeated BFS applies BFS once for each source vertex, so it has a time complexity of $O(|V|^2 \times |W| + |V| \times |E| \times |W|)$. As the number of edges typically dominates the number of vertices, let us simplify this to $O(|V| \times |E| \times |W|)$.

Looking now at memoised iterative deepening, each *step* of the algorithm involves performing a set union operation for every edge. The time complexity of bit-vector union is $O(|V| \div |W|)$, assuming a single machine instruction can logically-or two $|W|$-bit words in one go. Memoised iterative deepening must repeat this step until a fixed point is reached, which is $D$ times where $D$ is the network diameter, that is, the length of the longest shortest-path. This results in an overall time complexity of $O(D \times |E| \times |V| \div |W|)$.

In conclusion, the memoised iterative deepening implementation is faster than the repeated BFS implementation whenever the network diameter $D$ is less than $|W|^2$, where $|W|$ is the number of bits in an *Int*. Of course, this is a theoretical, worst-case analysis. How do the approaches compare in practice?

## 8 Run-time performance

Table 1 shows the performance of various implementations of memoised iterative deepening (MID) and breadth-first search (BFS) running on Intel Core i7-6770HQ. Graphs *g0* and *g1* are real graphs provided by our industrial partners whereas *g2*, *g3*, and *g4* are synthetic, but representative of larger graphs that our partners are also interested in.

The table includes performance measurements for both Haskell and C versions of the algorithms. It is clear that MID is significantly better than BFS on the graphs of interest. It is also clear that our Haskell version of BFS is very slow by comparison to the C version. We see two reasons for this: (1) the C version implements the visited set using a mutable array, permitting $O(1)$ insertion and membership checking; and (2) the C version implements the fringe lists using arrays, which is less demanding on the memory subsystem of a modern PC.

The performance of the C version of MID can be improved by around $2\times$ using Intel AVX vector instructions, processing more than 64 bits a time in the bit-vector union operation. With GHC support for AVX vectors, the same idea could be exploited in the Haskell version.

## 9 Conclusions

Whether using Haskell or C to compute the average shortest path, MID can provide remarkably better performance than repeated BFS on low-diameter graphs. The choice of MID is especially compelling in Haskell where a fast implementation using immutable data structures arises quite naturally. The magnitude of the performance difference highlights the importance of *algorithm exploration* during program development. Indeed, an often-cited strength of high-level languages such as Haskell is that the programmer can explore different algorithms more quickly.

The MID algorithm is one that we have come upon while thinking about the average shortest path problem, and Haskell has provided an excellent environment in which to understand both why it works and how it relates to other path-finding algorithms. Haskell has also served as a host for a very concise yet highly-efficient implementation.

## Acknowledgements and Open Access

This work was supported by EPSRC grant EP/N031768/1 (project POETS). All of the programs and benchmark graphs used in this article are available online at:

`https://github.com/POETSII/asp`

## References

A. Mokhov, A. de Gennaro, G. Tarawneh, J. Wray, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev, and A. Brown (2017). Language and Hardware Acceleration Backend for Graph Processing. Forum on specification & Design Languages.

Okasaki, C. and Gill, A. (1998). Fast Mergeable Integer Maps. Workshop on ML, pages 77–86.