1 Introduction

PageRank (PR) is an algorithm used by Google Search to rank web pages in their search engine results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

PageRank algorithm can be generalized to measure the importance of any type of recursive documents. It can be viewed as a node weight metric for complex networks including social networks, transportation networks, electricity networks, species networks, etc. The computing of PageRank is, therefore, a fundamental yet nontrivial problem.

In this project, we propose a parallel PageRank calculation program based on the MapReduce framework.

2 Problem Formulation

The PageRank algorithm simulates a random surfer traveling within a directed graph. Given the initial weight configuration of nodes, the algorithm outputs the probability (weight) distribution which represents the likelihood of a person randomly traveling through the edges will arrive at any particular node.

Now we formulate the Map/Reduce version of the PageRank problem.

The mapper receives the pair of node and pagerank as key, and the list of adjacent nodes as value. It maps those key-value pairs to either the pairs of node and pagerank increment or the pairs of node and list of adjacent nodes. The intermediate pairs are aggregated by key and fed to the reducers.

The reducer receives the pairs emitted by the mappers and aggregates the pagerank increments and calculates the updated pagerank value.

---


3 Implementation

3.1 Data Type Definitions

In this section, we would like to introduce some shared building blocks, the data types, upon which both our sequential and parallel solutions are implemented.

3.1.1 Nodes

Here we used String to represent a general node in the concerned graph. Intuitively, Nodes are a set of Node. An instance of Nodes might be {“a”, “b”, “c”}.

3.1.2 Edges

In our implementation, Edges are a map for which the key type is Node, while the value type is a list of nodes, representing the nodes connected to the corresponding key node. An instance of Edges might be {“a”: [“b”, “c”], “b”: [“c”]}.

3.1.3 Graph

The Graph data type represents a directed graph for whose nodes we would like to calculate the page rank values. The fields of this data type are,

- nodes. A set of all the nodes in this graph.
- inEdges. A map from a node to a list of nodes from which it is linked.
- outEdges. A map from a node to a list of nodes to which it links.

And there are some utility functions for this data type,

- parseLine :: Graph -> String -> Graph. Formulate an inEdge and an outEdge from the given String, add them to the given Graph, then return the newly constructed Graph. Each line of the input file should conform to the format ‘fromNode toNode’.
- fromContent :: String -> Graph. Given a file content, apply parseLine to every line of the file content to construct a Graph.
- fromFile :: String -> IO Graph. Given a file name, utilize fromContent to construct a IO Graph.

3.1.4 PageRank

A PageRank data type is a map from a Node to its current PageRankValue, which is a double in our case.

It also has some utility functions, such as the mapper and the reducer functions to compute the page rank values for a given graph with MapReduce, and also a sequential method to compute page rank. We will explain more about these utility functions in the following sections.
3.2 Sequential Solution

The function type is defined as \( \text{PageRank} \rightarrow \text{Graph} \rightarrow \text{Int} \rightarrow \text{Double} \rightarrow \text{PageRank} \). We can interpret it as, “given initial PageRank, the corresponding graph, a number of iterations to compute, and a damping factor, returns the resulting PageRank after those iterations of computation in a sequential way”.

Our sequential solution to compute the \( \text{PageRank} \) values for the next iteration works in this way,

\[
\begin{align*}
1 & \text{for each node } n \text{ in all the Nodes of the Graph do} \\
2 & \quad \text{pr} \_n \leftarrow 0 \\
3 & \quad \text{for each edge } (m, n) \text{ of } n\text{'s inEdges do} \\
4 & \quad \quad \text{num} \_\text{of_out} \_\text{nodes}_m \leftarrow \text{the number of nodes to which } m \text{ links} \\
5 & \quad \quad \text{pr} \_\text{previous}_m \leftarrow \text{the previous page rank value of } m \\
6 & \quad \quad \text{pr} \_\text{delta}_m \leftarrow \text{pr} \_\text{previous}_m / \text{num} \_\text{of_out} \_\text{nodes} \\
7 & \quad \quad \text{pr} \_n \leftarrow \text{pr} \_n + \text{pr} \_\text{delta}_m \\
8 & \quad \text{update the new page rank value of node } n \text{ in the new PageRank data} \\
9 & \text{end} \\
10 & \text{one iteration of computation is completed, return the updated PageRank data}
\end{align*}
\]

3.3 Parallel Solution with customized MapReduce

The function type is also defined as \( \text{PageRank} \rightarrow \text{Graph} \rightarrow \text{Int} \rightarrow \text{Double} \rightarrow \text{PageRank} \). And the interpretation is also similar, despite that this time the page rank values for the next iteration will be calculated in a parallel way.

The function type of the mapper is defined as \( \text{mapper} :: (\text{PageRankValue}, [\text{Node}]) \rightarrow \text{PageRank} \). For each \( \text{Node} \) in the \( \text{Graph} \), the mapper takes its current \( \text{PageRankValue} \) and the list of nodes in its \( \text{outEdges} \), then produces a map for which the key is each of the node in its \( \text{outEdges} \), and the value is its contribution to that node, defined as its current \( \text{PageRankValue} \) divided by the number of nodes in its \( \text{outEdges} \).

The function type of the reducer is defined as \( \text{reducer} :: [\text{PageRank}] \rightarrow \text{PageRank} \). The reducer merges all the outputs that the mapper produces. The merging rule is a simple addition for each same node.

With these definitions, our customized mapReduce function is implemented as,

\[
\begin{align*}
1 & \text{mapReduce} :: (a \rightarrow b) \rightarrow ([b] \rightarrow c) \rightarrow [a] \rightarrow c \\
2 & \text{mapReduce \_mapper \_reducer \_input} = \text{pseq mapResult \_reduceResult} \\
3 & \quad \text{where} \\
4 & \quad \text{mapResult} = \text{parMap \_rpar \_mapper \_input} \\
5 & \quad \text{reduceResult} = \text{runEval (rpar \$ reducer \_mapResult)}
\end{align*}
\]

Once the reducer completed its work in one iteration, we could simply update the page rank value for each node as \((\text{base} + d \times \text{pr})\), where \(\text{base} = (1 - d) / \text{num} \_\text{of_nodes} \_\text{in_graph} \), \(d\) is the damping factor, \(\text{pr}\) is the corresponding value the reducer produced.
3.4 Benchmark based on External MapReduce Library

We wanted to have an external benchmark with which to compare and evaluate our MapReduce based parallel PageRank implementation.

A short web search yielded Haskell-MapReduce (https://github.com/jdstmporter/Haskell-MapReduce, https://wiki.haskell.org/MapReduce_as_a_monad) to be a promising general-purposed MapReduce library. Therefore we implemented a benchmark based on the mentioned library.

The library is implemented in a monadic fashion such that mappers and reducers can be viewed as generalized transformers of type signature \( a \rightarrow ([s, a]) \rightarrow ([s', b]) \). It provides a wrapper function \( \text{liftMR} \) that converts the map / reduce function into a monadic function.

Given the aforementioned MapReduce library, we only need to implement conventional mapper and reducer.

According to the specification of the library, mapper should take the form of \([s] \rightarrow ([s', a])\), where \( s \) is input data, \( s' \) is output data and \( a \) is output key. We implemented the mapper such that \( s = (\text{fromNode}, (\text{pageRankValue}, \text{toNodes})) \) and \( s' = (\text{toNode}, \text{pageRankIncrement}) \). Each of the input data emits its pagerank increment contribution to all of its \text{toNodes}.

The reducer is implemented in a similar fashion, it takes input of the form \(([\text{toNode}, \text{pageRankIncrement}])\). For a particular \( \text{toNode} \), the pagerank increment contribution from all \( \text{fromNodes} \) are aggregated together, producing the pagerank value.

The evaluations to be given later in this report showed that this external benchmark has a vastly worse performance compared with our implementation.

4 Evaluation

4.1 Settings

We performed our experiments on a MacBook Pro (15-inch, 2018), of which the processor is 2.2 GHz 6-core Intel Core i7, and the memory is 16 GB 2400 MHz DDR4.

4.2 Experiment Results

We performed our experiments by performing 10 iterations of page rank computation on two datasets with different sizes.

The first dataset is a larger fraction of the Wikipedia Note Network\(^4\) which is 90Kb large with 11515 edges. Table\(^1\) shows the experiment results of our MapReduce implementation. Table\(^2\) shows the experiment results of our sequential implementation and the benchmark implementation.

\(^3\)https://github.com/jdstmporter/Haskell-MapReduce
\(^4\)https://snap.stanford.edu/data/wiki-Vote.html
### Table 1: Experiment Result for a 90Kb Dataset (MapReduce)

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
<th>converted</th>
<th>gc’d</th>
<th>fizzled</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59.92</td>
<td>0</td>
<td>4412</td>
<td>33038</td>
<td>37450</td>
</tr>
<tr>
<td>2</td>
<td>37.68</td>
<td>23322</td>
<td>858</td>
<td>13270</td>
<td>37450</td>
</tr>
<tr>
<td>3</td>
<td>35.86</td>
<td>28519</td>
<td>535</td>
<td>8396</td>
<td>37450</td>
</tr>
<tr>
<td>4</td>
<td>35.8</td>
<td>30830</td>
<td>346</td>
<td>4757</td>
<td>37450</td>
</tr>
<tr>
<td>5</td>
<td>36.09</td>
<td>32348</td>
<td>346</td>
<td>4757</td>
<td>37450</td>
</tr>
<tr>
<td>6</td>
<td>34.13</td>
<td>33376</td>
<td>300</td>
<td>3774</td>
<td>37450</td>
</tr>
<tr>
<td>7</td>
<td>35.62</td>
<td>33595</td>
<td>295</td>
<td>3560</td>
<td>37450</td>
</tr>
<tr>
<td>8</td>
<td>38.03</td>
<td>34186</td>
<td>256</td>
<td>3008</td>
<td>37450</td>
</tr>
<tr>
<td>9</td>
<td>40.4</td>
<td>34642</td>
<td>243</td>
<td>2565</td>
<td>37450</td>
</tr>
<tr>
<td>10</td>
<td>44.66</td>
<td>34850</td>
<td>226</td>
<td>2374</td>
<td>37450</td>
</tr>
<tr>
<td>11</td>
<td>44.29</td>
<td>35192</td>
<td>212</td>
<td>2046</td>
<td>37450</td>
</tr>
<tr>
<td>12</td>
<td>48.84</td>
<td>35487</td>
<td>191</td>
<td>1772</td>
<td>37450</td>
</tr>
</tbody>
</table>

### Table 2: Experiment Result for a 90Kb Dataset (Sequential & Benchmark)

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>173.62</td>
</tr>
<tr>
<td>benchmark-1</td>
<td>1923.26</td>
</tr>
<tr>
<td>benchmark-6</td>
<td>906.79</td>
</tr>
</tbody>
</table>

The second dataset is a smaller fraction of the Wikipedia Note Network, which is 40Kb large with 5508 edges. Table 3 shows the experiment results of our MapReduce implementation. Table 4 shows the experiment results of our sequential implementation and the benchmark implementation.

### Table 3: Experiment Result for a 40Kb Dataset (MapReduce)

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
<th>converted</th>
<th>gc’d</th>
<th>fizzled</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.23</td>
<td>0</td>
<td>2371</td>
<td>24879</td>
<td>27250</td>
</tr>
<tr>
<td>2</td>
<td>21.96</td>
<td>15890</td>
<td>844</td>
<td>10516</td>
<td>27250</td>
</tr>
<tr>
<td>4</td>
<td>20.38</td>
<td>21898</td>
<td>407</td>
<td>4945</td>
<td>27250</td>
</tr>
<tr>
<td>6</td>
<td>19.71</td>
<td>23888</td>
<td>286</td>
<td>3076</td>
<td>27250</td>
</tr>
<tr>
<td>8</td>
<td>24.64</td>
<td>24641</td>
<td>238</td>
<td>2371</td>
<td>27250</td>
</tr>
<tr>
<td>10</td>
<td>26.32</td>
<td>25345</td>
<td>196</td>
<td>1709</td>
<td>27250</td>
</tr>
<tr>
<td>12</td>
<td>27.99</td>
<td>25883</td>
<td>141</td>
<td>1226</td>
<td>27250</td>
</tr>
</tbody>
</table>

### 4.3 Performance Analysis

From the results, we can conclude that our MapReduce implementation is much more efficient both than the sequential version and than the benchmark implementation.
Table 4: Experiment Result for a 40Kb Dataset (Sequential & Benchmark)

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>88.26</td>
</tr>
<tr>
<td>benchmark-1</td>
<td>804.94</td>
</tr>
<tr>
<td>benchmark-6</td>
<td>472.71</td>
</tr>
</tbody>
</table>

We can also observe that when $N = 6$, which is equal to the number of cores, the performance of our implementation is the best. If $N$ is set to be larger, even the conversion rate is increased, the overhead for parallelism is also increased, hence the consumed time becomes longer.

For further analysis, we scrutinized the event log for our MapReduce implementation running with the 40Kb dataset using *ThreadScope*. From the figure, we can observe that the bottleneck is the GC waiting time.

![Eventlog for MapReduce experiment with 40Kb Dataset](image)

Figure 1: Eventlog for MapReduce experiment with 40Kb Dataset

A  Code Listing
The main application program

Command line arguments: `infilePath`, `outfilePath`, `itrs`, `[mode]`

`infilePath`: path of input file, which should be in the format of lines consisting of `fromNode toNode`

`outfilePath`: path of output file

`itrs`: number of iterations in the PageRank computation

`mode`: optional, mode of PageRank computation, one of `{seq, mr_def, mr_ext}`, default to `mr_def`

`seq`: non-parallel sequential computation

`mr_def`: parallel implementation based on default MapReduce

`mr_ext`: benchmark parallel implementation based on external opensourced MapReduce library

module Main (main) where

import Control.Monad (when)
import System.IO (openFile, IOMode(WriteMode), hPutStrLn, hClose)
import System.Environment (getArgs, getProgName)
import System.Exit
import Data.Map as M (toList)
import ProcessData (processData)
import PageRank (computePageRankSeq, computePageRankMR)
import PageRankExt (computePageRankMRext)

main :: IO()
main = do
  progName <- getProgName
  args <- getArgs

  when (length args /= 3 && length args /= 4) $ do
    die $ "Usage: " ++ progName ++ " <infilePath> <outfilePath> <itrs> [mode], where\n    \ mode is one of \{seq, mr_def, mr_ext\}, default to mr_def"

  let `infilePath` : `outfilePath` : `itrs` : `mode` = args
  computePageRank = case `mode` of
    [] -> computePageRankMR
    ["mr_def"] -> computePageRankMR
    ["seq"] -> computePageRankSeq
    ["mr_ext"] -> computePageRankMRext
    _ -> error $ "Usage: " ++ progName ++ " <infilePath> <outfilePath> <itrs> [mode], where\n    \ mode is one of \{seq, mr_def, mr_ext\}, default to mr_def"
Listing 1: app/Main.hs

```haskell
(graph, pageRank) <- processData filePath
let resPageRank = computePageRank pageRank graph (read itrs) 0.85
h <- openFile outFile WriteMode
mapM_ (hPutStrLn h) [ n ++ ": " ++ show pr | (n, pr) <- M.toList resPageRank ]
hClose h
```

Listing 2: src/ProcessData.hs

```haskell
module ProcessData
(processData) where

import Graph (Graph, fromFile)
import PageRank (PageRank, initFromGraph)

processData :: String -> IO (Graph, PageRank)
processData filename = do
  graph <- fromFile filename
  let pageRank = initFromGraph graph
  return (graph, pageRank)
```

Listing 3: src/Graph.hs

```haskell
module Graph
(Graph(..), Node)
```

This module contains a utility function, which
1) reads in a graph from a input file
2) initializes a PageRank data from the given graph
3) returns the graph and the initial page rank

This module defines the Graph data type. The fields are,
1) nodes: set of all the nodes in this graph
2) inEdges: map from a node to a list of nodes from which it is linked
3) outEdges: map from a node to a list of nodes to which it links

This module also contains some utility functions for this data type.
import qualified Data.Map as M (Map, insertWith, empty, keysSet)
import qualified Data.Set as S (Set, union, fromList, empty, toList, difference)
import System.IO (readFile)

type Node = String

fromFile :: String -> IO Graph
fromFile filename = do
  content ← readFile filename
  return $ fromContent content

fromContent :: String -> Graph
fromContent content =
  let ls = lines content
  in postProcess $ foldl parseLine empty ls
    where
      postProcess :: Graph -> Graph
      postProcess graph = foldl parseLine graph newLines
        where
          ns = nodes graph
          sinkNodes = S.difference ns $ M.keysSet $ outEdges graph
          newLines = [ n1 ++ " " ++ n2 |
                       n1 ← S.toList sinkNodes, n2 ← S.toList ns, n1 /= n2 ]

parseLine :: Graph -> String -> Graph
parseLine graph line =
  let ws = words line
    in case ws of
[fromNode, toNode] ->
  Graph ns iEdges oEdges
  where
    ns = S.union (S.fromList ws) (nodes graph)
    iEdges = M.insertWith (++) toNode [fromNode] (inEdges graph)
    oEdges = M.insertWith (++) fromNode [toNode] (outEdges graph)

Listing 3: src/Graph.hs

module MapReduce
  ( mapReduce )
where

import Control.Parallel (pseq)
import Control.Parallel.Strategies (rpar, runEval, parMap)

mapReduce ::
  (a -> b) -- map function
  -> ([b] -> c) -- reduce function
  -> [a]
  -> c
mapReduce mapper reducer input = pseq mapResult reduceResult
  where mapResult = parMap rpar mapper input
        reduceResult = runEval (rpar $ reducer mapResult)

Listing 4: src/MapReduce.hs

{-

This module defines the PageRank data type, which is a map from a node to its
current page rank value.

This module also contains the empty definition and some utility functions for
this data type, such as the mapper and the reducer functions to compute the page
rank values for a given graph with MapReduce, and also a sequential method to
compute page rank.

-}

module PageRank
  ( PageRank, initFromGraph, computePageRankSeq

10
```haskell
import Graph (Graph(..), Node)
import MapReduce (mapReduce)
import qualified Data.Map as M (Map, empty, fromList, lookup, unionWith, toList)
import qualified Data.Set as S (toList, size)
import Data.Maybe (fromJust)

type PageRankValue = Double

-- Initial state of a PageRank data for an empty graph
empty :: PageRank
empty = M.empty

{-
  Initial state of a PageRank data for a given graph, the page rank value
  of each node is the reciprocal of the number of nodes in this graph
-}
initFromGraph :: Graph -> PageRank
initFromGraph graph =
  let ns = nodes graph
      pr = 1.0 / (fromIntegral $ S.size ns) in
  M.fromList [ (n, pr) | n <- S.toList ns ]

mapper :: (PageRankValue, [Node]) -> PageRank
mapper (pr, outNodes) =
  let pr' = pr / (fromIntegral $ length outNodes) in
  M.fromList [ (n, pr') | n <- outNodes ]

reducer :: [PageRank] -> PageRank
reducer [] = empty
reducer [x] = x
reducer (x:xs) = M.unionWith (+) x (reducer xs)

{-
  Given initial PageRank and the corresponding graph, a number of iterations
  to compute, and a damping factor, returns the resulting PageRank after those
  iterations of computation in a parallel way with MapReduce
-}
computePageRankMR :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankMR pageRank graph itrs damping =
  let nextPageRank = computeNextPageRankMR pageRank
  in computePageRankMR nextPageRank graph (itrs-1) damping
  where
```
computeNextPageRankMR :: PageRank -> PageRank
computeNextPageRankMR curPR = 
  let ns = S.toList $ nodes graph 
      input = map produceInput ns 
      produceInput n = (pr, outNodes) 
    where 
      pr = fromJust $ M.lookup n curPR 
      outNodes = fromJust $ M.lookup n $ outEdges graph 
      mrResult = mapReduce mapper reducer input 
      base = (1 - damping) / (fromIntegral $ length ns) 
    in M.fromList [ (n, base + damping * pr) | (n, pr) <- M.toList mrResult ]

{- Given initial PageRank and the corresponding graph, a number of iterations 
to compute, and a damping factor, returns the resulting PageRank after those 
iterations of computation in a sequential way -}
computePageRankSeq :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankSeq pageRank _ 0 _ = pageRank
computePageRankSeq nextPageRank graph (itrs-1) damping = 
  let nextPageRank = computeNextPageRank pageRank 
    in computePageRankSeq nextPageRank graph (itrs-1) damping 
  where 
    computeNextPageRank :: PageRank -> PageRank
    computeNextPageRank curPR = 
      M.fromList [ (n, computePRValue n) | n <- ns ] 
    where 
      ns = S.toList $ nodes graph 
      iEdges = inEdges graph 
      oEdges = outEdges graph 
      computePRValue :: Node -> PageRankValue 
      computePRValue n = 
        let inNodes = fromJust $ M.lookup n iEdges 
            sumUp acc node = 
              let numOutNodes = length $ fromJust $ M.lookup node oEdges 
                  prValue = fromJust $ M.lookup node curPR 
                in acc + prValue / (fromIntegral numOutNodes) 
        in (1 - damping) / (fromIntegral $ length ns) + damping * (foldl sumUp 0 inNodes)
Module that defines the 'MapReduce' monad and exports the necessary functions.

Mapper / reducers are generalised to functions of type
@*\textit{a} \rightarrow \{(s,a) \rightarrow (s',b)\}@ which are combined using the monad's bind operation. The resulting monad is executed on initial data by invoking 'runMapReduce'.

For programmers only wishing to write conventional map / reduce algorithms, which use functions of type @*\textit{a} \rightarrow \{(s) \rightarrow (s',b)\}@ a wrapper function 'liftMR' is provided, which converts such a function into the appropriate monadic function.

module MapReduceLibExt (
    * Types
    MapReduce,
    * Functions
    *
    ** Monadic operations
    return, (>>=),
    ** Helper functions
    run, distribute, lift ) where

import Data.List (nub)
import Control.Applicative ((<$>))
import Control.Monad (liftM)
import Control.DeepSeq (NFData)
import System.IO
import Prelude hiding (return,(>>=))
import Data.Digest.Pure.MD5
import Data.Binary
import qualified Data.ByteString.Lazy as B
import Control.Parallel.Strategies (parMap, rdeepseq)

The parallel map function; it must be functionally identical to 'map', distributing the computation across all available nodes in some way.

\[ \text{pMap} :: (NFData \textit{b}) \Rightarrow (\textit{a} \rightarrow \textit{b}) \rightarrow [\textit{a}] \rightarrow [\textit{b}] \]

Generalised version of 'Monad' which depends on a pair of 'Tuple's, both of which change when '>>=' is applied.

class MonadG \textit{m} where
    return :: \textit{a} \rightarrow \textit{m} \textit{s} \times \textit{s} \textit{a} -- `value. transformation that inserts the value by replacing all
The key values with the specified value, leaving the data unchanged.

\[
(\gg\gg) :: (\text{Eq } b, \text{NFData } s'', \text{NFData } c) \Rightarrow \\
\text{m s a s' b} \quad \rightarrow \quad \text{Initial processing chain} \\
\text{m s a s' c} \quad \rightarrow \quad \text{Extended processing chain}
\]

The basic type that provides the MapReduce monad (strictly a generalised monad).

In the definition \((s,a)\) is the type of the entries in the list of input data and \((s',b)\) that of the entries in the list of output data, where \(s\) and \(s'\) are data and \(a\) and \(b\) are keys.

'MapReduce' represents the transformation applied to data by one or more MapReduce staged. Input data has type \(\{(s,a)\}\) and output data has type \(\{(s',b)\}\) where \(s\) and \(s'\) are data types and \(a\), \(b\) are key types.

Its structure is intentionally opaque to application programmers.

newtype MapReduce s a s' b = MR \{ runMR :: \{(s,a)\} \rightarrow \{(s',b)\} \}

Make MapReduce into a 'MonadG' instance

instance MonadG MapReduce where

\(\gg\gg\) = bind

Insert a value into 'MapReduce' by replacing all the key values with the specified value, leaving the data unchanged.

\[
\text{ret} :: a \rightarrow \text{MapReduce } s x s a \\
\text{ret k} = \text{MR (\{ss \rightarrow \{(s,k) | s <\text{fst }<\gg ss\}\})}
\]

Apply a generalised mapper / reducer to the end of a chain of processing operations to extend the chain.

\[
\text{bind} :: (\text{Eq } b, \text{NFData } s'', \text{NFData } c) \Rightarrow \\
\text{MapReduce } s a s' b \quad \rightarrow \quad \text{Initial state of the monad} \\
\text{b} \rightarrow \text{MapReduce } s' b s'' c \quad \rightarrow \quad \text{Transformation to append to it} \\
\text{MapReduce } s a s'' c \quad \rightarrow \quad \text{Extended transformation chain}
\]

\[
\text{bind } f g = \text{MR (\{s \rightarrow \}} \\
\text{let} \\
\text{fs} = \text{runMR } f s \\
\text{gs} = \text{map } g \$ \text{nub } s \text{ snd } <\gg fs
\]
in concat $ pMap ("runMR" fs) gs

-- | Execute a MapReduce MonadG given specified initial data. Therefore, given
-- a 'MapReduce' @m@ and initial data @xs@ we apply the processing represented
-- by @m@ to @xs@ by executing
--
-- @run m xs@

run :: MapReduce s () s' b
  -- ^ 'MapReduce' representing the required processing
  -> [s]
  -- ^ Initial data
  -> [(s',b)]
  -- ^ Result of applying the processing to the data
run m ss = runMR m [(s,()) | s <- ss]

-- | The hash_ function. Computes the MD5 hash_ of any 'Hashable' type
hash_ :: (Binary s) => s
  -- ^ The value to hash_
  -> Int
  -- ^ its hash_
hash_ s = sum $ map fromIntegral (B.unpack h)
  where
    h = encode (md5 $ encode s)

-- | Function used at the start of processing to determine how many threads of processing
-- to use. Should be used as the starting point for building a 'MapReduce'.
--
-- Therefore a generic 'MapReduce' should look like
--
-- @' distribute ' 'f1 ' '

distribute :: (Binary s) => Int
  -- ^ Number of threads across which to distribute initial data
  -> MapReduce s () s Int
  -- ^ The 'MapReduce' required to do this
distribute n = MR ($ ss -> [(s,hash_ s 'mod' n) | s <- fst <$> ss])

-- | The wrapper function that lifts mappers / reducers into the 'MapReduce'
-- monad. Application programmers can use this to apply MapReduce transparently
-- to their mappers / reducers without needing to know any details of the implementation
-- of MapReduce.
--
-- Therefore the generic 'MapReduce' using only traditional mappers and
-- reducers should look like

-- @' distribute ' 'lift ' 'f1 ' '

lift :: (Eq a) => ([s] -> [(s',b)])
  -- ^ traditional mapper / reducer of signature
  -> @([s] -> [(s',b)])@
  -- ^ the input key
  -> MapReduce s a s' b
  -- ^ the mapper / reducer wrapped as an instance
  -- ^ of 'MapReduce'
lift f k = MR ($ ss -> f <$> filter ($ s -> k == snd s) ss)

Listing 6: src/MapReduceLibExt.hs
The benchmark PageRank computation implementation based on external opensourced MapReduce library GitHub repository of the MapReduce library: https://github.com/jdstmporter/Haskell–MapReduce

module PageRankExt
(
    computePageRankMRext
) where

import Graph (Graph(..), Node)
import qualified Data.Map as M (Map, empty, fromList, lookup, unionWith, toList)
import qualified Data.Set as S (toList, size)
import Data.Maybe (fromJust)
import MapReduceLibExt (run, distribute, lift, (>==))

type PageRankValue = Double

type PageRank = M.Map Node PageRankValue

empty :: PageRank
empty = M.empty

initFromGraph :: Graph -> PageRank
initFromGraph graph = let ns = nodes graph
pr = 1.0 / (fromIntegral $ S.size ns) in M.fromList [(n, pr) | n <- S.toList ns]

mr :: Double -> Double -> Int -> 
[(Node, (PageRankValue, [Node]))] -> 
[(Node, (PageRankValue, [Node]))] -> run f state
where
f = distribute n MapReduceLibExt.>== lift mapper MapReduceLibExt.>== lift (reducer damping numNodes)

mapper :: 
[(Node, (PageRankValue, [Node]))] -> 
[((Node, (PageRankValue, [Node])), Node)]
mapper [] = []
mapper (x:xs) = parse x ++ mapper xs
where
parse (n, (pr, outNodes)) = let pr' = pr / (fromIntegral $ length outNodes)
in [(n, (0, outNodes)), n] : [(n, (pr', [])), n] | n <- outNodes]

reducer :: [(Node, (PageRankValue, [Node]))] -> [(Node, (PageRankValue, [Node]))]
reducer [] = []
reducer (x:xs) = reduce x ++ reducer xs
where
reduce (n, (pr, outNodes)) = let n' = n / (fromIntegral $ length outNodes)
in [(n, (0, outNodes)), n] : [(n, (pr, [])), n] | n <- outNodes]

According to the specification of Haskell–MapReduce lib
mapper should take the form of s -> [s', a]
where s is input data, s' is output data and a is output key
mapper :: 
[(Node, (PageRankValue, [Node]))] -> 
[((Node, (PageRankValue, [Node])), Node)]
mapper [] = []
mapper (x:xs) = parse x ++ mapper xs
where
parse (n, (pr, outNodes)) = let pr' = pr / (fromIntegral $ length outNodes)
in [(n, (0, outNodes)), n] : [(n, (pr', [])), n] | n <- outNodes]

reducer should take the form of s' -> s''
where s' is output data of mapper, s'' is output data of reducer
reducer :: Double -> Double -> [[(Node, (PageRankValue, [Node]))]] -> [[(Node, (PageRankValue, [Node]))]]
reducer _ _ [] = []
reducer damping numNodes xs@(_:_) =
  [ foldl f (fst x, ((1 - damping) / numNodes, [])) xs ]
  where f x y = (fst x, (
    (fst $ snd x) + damping * (fst $ snd y),
    (snd $ snd x) ++ (snd $ snd y)
  )
)

computePageRankMRext :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankMRext pageRank _ 0 _ = pageRank
computePageRankMRext pageRank graph itrs damping =
  let ns = S.toList $ nodes graph
      numNodes = fromIntegral $ length ns
      oEdges = outEdges graph
      initMRinput = map toMRinput ns
      toMRinput n =
        let pr = fromJust $ M.lookup n pageRank
            outNodes = fromJust $ M.lookup n oEdges
            in (n, (pr, outNodes))
      inM rOutput = mrlr initMRinput damping numNodes itrs
      inM fromList [ (n, pr) | (n, (pr, _)) <- mrOutput ]
  where
    mrlr :: [(Node, (PageRankValue, [Node]))] -> Double -> Double -> Int -> [(Node, (PageRankValue, [Node]))]
    mrlr input _ _ 0 = input
    mrlr input damping numNodes itrs =
      let output = mr damping numNodes 1 input
          in mrlr output damping numNodes (itrs-1)

import Data.Map as M (toList)
import ProcessData (processData)
import PageRank (computePageRankMR)

main :: IO ()
main = do
  (graph, pageRank) <- processData "../data/sample_input.txt"
  let resPageRank = computePageRankMRext pageRank graph 10 0.85
      mapM_ putStrLn $ map (n ++ ": " ++ show pr | (n, pr) <- M.toList resPageRank )

Listing 7: src/PageRankExt.hs

Listing 8: test/Spec.hs