MapReduce formulation of PageRank

Maarten Fokkinga


For the theoretical discussion, we assume the following be given:

A graph, henceforth called ‘the graph’

Variables \( m \) and \( n \) range over the nodes of the graph

\[ \text{out } m = \text{a bag of out links (each one represented by the target node) from node } m \]

\[ C_m = \text{the cardinality of } \text{out } m = \# \text{out } m \]

**PageRank.** To simplify the formulas, we use an auxiliary function (for a global constant \( d \)):

\[
demp x = \frac{1 - d}{N} + d \times x
\]

Here, \( N \) equals the number of nodes in the graph for which a page rank will be computed; some of those nodes might appear in the links but not be given explicitly. Using the \( d\text{emp} \) function, the given PageRank specification reads as follows:

\[
PR_m = demp \left( \frac{PR_{n_1}}{C_{n_1}} + \cdots + \frac{PR_{n_k}}{C_{n_k}} \right)
\]

where \( n_1, \ldots, n_k \) are the nodes that have \( m \) in their out links.

**Remark.** The equation is of the form ‘\( x = \ldots x \ldots \)’ (where \( x \) ranges over functions), or even ‘\( x = f x \)’, when we capture the dots in a function \( f \). We bypass the mathematical theory under what conditions such an equation has a solution for \( x \), and under what conditions the equation has several solutions of which one is the “obviously intended” one. We simply assume that a solution exists and that the desired solution has the form of, and can be computed as, the limit of \( x_0, f x_0, f(f x_0), \ldots \), for a suitable initial value \( x_0 \). We furthermore assume that the limit has been sufficiently approximated when two successive terms in the chain are “near” to each other. This is called “the iterative approach”.

**Remark.** An additional constraint is that \( \sum_{m \in \text{nodes}} PR_m \) equals 1; that is, \( PR \) is a probability distribution. This constraint is consistent with the equation but doesn’t seem to follow from the equation.

**Remark.** It is allowed that \( \text{out } n \) is a bag rather than a set, that is, node \( m \) occurs multiple times in the out links of node \( n \). In that case, \( n \) occurs as many times among \( n_1, \ldots, n_k \) as the multiplicity of \( m \) in \( \text{out } n \). If the corresponding effect on the page rank is undesirable, then one should take care that each \( \text{out } n \) is a set.

Taking a “reasonable” \( PR_0 \), we assume that \( PR \) is the limit of \( PR_0, PR_1, PR_2, \ldots \), where \( PR_{i+1} \) is expressed in terms of \( PR_i \) according to the page rank specification:

\[
PR_{i+1} m = \text{demp} \left( \sum_{n \mid m \in \text{out } n} \frac{PR_i n}{C n} \right)
\]
Iteration step. The computation of $PR_{i+1}$ from $PR_i$ can be easily formulated as a MapReduce computation once we realize that all $PR_{i+1}$-values can be computed simultaneously from all $PR_i$-values. To show this by a concrete example, suppose we have nodes $a$, $b$, $c$ with out-links $out\ a = \{a, c\}$, $out\ b = \{c\}$, $out\ c = \{a, b, c\}$. To express $PR_{i+1}$ in terms of $PR_i$, first rewrite each $PR_i\ m$ as a summation of $k$ times $\frac{PR_i\ m}{k}$, where $k = C\ m$:

\[
\begin{array}{ccc}
\ a & \ b & \ c \\
PR_i\ a & = & \frac{PR_i\ a}{2} + \frac{PR_i\ a}{2} \quad \text{recall: out}\ a = \{a, c\} \text{ and } C\ a = 2 \\
PR_i\ b & = & \frac{PR_i\ b}{1} \quad \text{recall: out}\ b = \{c\} \text{ and } C\ b = 1 \\
PR_i\ c & = & \frac{PR_i\ c}{3} + \frac{PR_i\ c}{3} + \frac{PR_i\ c}{3} \quad \text{recall: out}\ c = \{a, b, c\} \text{ and } C\ c = 3 \\
\end{array}
\]

Now, a summation and subsequent demping of the columns gives the $PR_{i+1}$-values:

\[
\begin{align*}
PR_{i+1}\ a & = \text{demp ("sum of column under } a") \\
PR_{i+1}\ b & = \text{demp ("sum of column under } b") \\
PR_{i+1}\ c & = \text{demp ("sum of column under } c") \\
\end{align*}
\]

The partitioning of $a$’s $PR$-value into the two terms under columns $a, c$ (and similarly for $b$ and $c$) can be done by a mapper. If, in addition, the column name is added as a key to the terms, then the grouping of the terms per column name is done by the MapReduce framework. Finally, the summation and demping of terms with equal keys (i.e., column names) can be done by a reducer. For our example, the three MapReduce phases read as follows:

The mapper’s action:

\[
\begin{align*}
(a, PR_i\ a) & \xrightarrow{\text{mapper}} (a, \frac{PR_i\ a}{C\ a}), \quad (c, \frac{PR_i\ a}{C\ a}) \\
(b, PR_i\ b) & \xrightarrow{\text{mapper}} (c, \frac{PR_i\ b}{C\ b}) \\
(c, PR_i\ c) & \xrightarrow{\text{mapper}} (a, \frac{PR_i\ c}{C\ c}), \quad (b, \frac{PR_i\ c}{C\ c}), \quad (c, \frac{PR_i\ c}{C\ c}) \\
\end{align*}
\]

The grouping of the mapper results gives:

\[
\begin{align*}
(a, \left[\frac{PR_i\ a}{C\ a}, \frac{PR_i\ c}{C\ c}\right]), \\
(b, \left[\frac{PR_i\ c}{C\ c}\right]), \\
(c, \left[\frac{PR_i\ a}{C\ a}, \frac{PR_i\ b}{C\ b}, \frac{PR_i\ c}{C\ c}\right]) \\
\end{align*}
\]

The reducer’s action:

\[
\begin{align*}
(a, \left[\frac{PR_i\ a}{C\ a}, \frac{PR_i\ c}{C\ c}\right]) & \xrightarrow{\text{reducer}} (a, \text{demp (}\frac{PR_i\ a}{C\ a} + \frac{PR_i\ c}{C\ c})) \\
(b, \left[\frac{PR_i\ c}{C\ c}\right]) & \xrightarrow{\text{reducer}} (b, \text{demp (}\frac{PR_i\ c}{C\ c})) \\
(c, \left[\frac{PR_i\ a}{C\ a}, \frac{PR_i\ b}{C\ b}, \frac{PR_i\ c}{C\ c}\right]) & \xrightarrow{\text{reducer}} (c, \text{demp (}\frac{PR_i\ a}{C\ a} + \frac{PR_i\ b}{C\ b} + \frac{PR_i\ c}{C\ c})) \\
\end{align*}
\]

Notice that, when representing of a function by the set of its (input, output)-pairs, the set of inputs for the mapper equals $PR_i$ and the set of outputs from the reducer equals $PR_{i+1}$. (The above exposition is really convincing, I think, but the appendix gives a fully formal proof and even derivation.) Thus, representing $out\ m$ by a list, we have:

\[
\text{mapper (}\ m, p) = [(n, \frac{p}{C\ m}) \mid n \leftarrow out\ m]
\]
reducer \((m, ps)\) = \((m, \text{demp} (\text{sum} \, ps))\)

and

\[ step = \text{mapReduce mapper reducer} \]

\[ PR_{i+1} = \text{step} \, PR_i \]

-- viewing \(PR_j\) as a set of (input, output)-pairs

Adaptations. One adaptation (and deviation from the specification!) is necessary in order to guarantee that \(PR_{i+1}\) is a probability distribution again. If \(out \, m\) is empty, then the value \(PR_i \, m\) vanishes in the iteration step so that the \(PR_{i+1}\)-values no longer sum up to 1. For a node with no out links, we may decide that its probability mass is assigned to the node itself (an alternative would be to distribute the probability mass over, say, all nodes of the graph). Thus the adapted mapper reads as follows:

\[ \text{mapper} \, (m, p) = \left[ \left( n, \frac{p}{\text{out} \, m} \right) \mid n \leftarrow \text{out} \, m \right] + \left[ \left( m, p \right) \mid \text{out} \, m = [] \right] \]

Moreover, another adaptation is needed if the MapReduce computation also has to keep the outgoing links locally available for each node so that there is no need for a globally available graph. To achieve this, we assume that each mapper gets the pair \((m, (p, \text{out} \, m))\) instead of just \((m, p)\), and, to make sure that the reducer can construct such pairs again, the mapper yields suitable outgoing links in addition to the \(PR\)-values:

\[ \text{mapper} \, (m, (p, ns)) = \left[ \left( n, \left( \frac{p}{\text{out} \, ns} \right), \emptyset \right) \mid n \leftarrow ns \right] + \left[ \left( m, (p, \emptyset) \right) \mid ns = [] \right] + \left[ \left( m, (0, ns) \right) \mid ns \neq [] \right] \]

\[ \text{reducer} \, (m, xs) = \left( m, \left( \text{demp} (\text{sum} \, ps), \text{concat} \, ns \right) \right) \]

where \(ps = \text{map} \, \text{fst} \, xs\), \(ns = \text{map} \, \text{snd} \, xs\)

Iteration. To iterate the step from \(PR_i\) to \(PR_{i+1}\), first define when two functions with the same domain are “near” to each other; say, for a given constant \(\text{epsilon}\):

\[ \text{near} \, f \, g = \max[\text{abs} (f \, x - g \, x) \mid x \leftarrow \text{domain} \, f] < \text{epsilon} \]

In our case, the \(PR\) functions are represented as (input, output)-pairs where each output itself is a (probability, out links)-pair, so that:

\[ \text{near} \, pr_1 \, pr_2 = \max[\text{abs} (p_1 - p_2) \mid (n_1, (p_1, ns_1)) \leftarrow pr_1; (n_2, (p_2, ns_2)) \leftarrow pr_2; n_2 = n_1] < \text{epsilon} \]

Second, iterate the step until two successive items are near to each other:

\[ \text{iterate} \, f \, x_0 = x_0 : \text{iterate} \, f \, (f \, x_0) \mid = [x_0, f \, x_0, f(f \, x_0), f(f(f \, x_0)), \ldots] \]

\[ \text{limit} \, c \, (x : y : zs) = x, \text{if} \, c \, x \, y \]

\[ = \text{limit} \, c \, (y : zs), \text{otherwise} \]

\[ \text{the} \, PR = (\text{limit} \, \text{near} \, \text{. iterate} \, \text{step}) \, \text{init} \, PR \]

The definition of \(\text{init} \, PR\) heavily depends on the way the data is given. Let the graph be given as a collection of \((m, \text{out} \, m)\)-pairs. If we want \(\text{init} \, PR\) to be a flat probability distribution over all nodes, then we have:

\[ \text{init} \, PR = \text{mapReduce} \, f \, \text{id} \, \, \text{“the graph”} \quad \text{where} \, f \, (m, ns) = (m, (\frac{1}{N}, ns)) \]
Appendix

We formally prove that $PR_{i+1} = mapReduce \ mapper \ reducer \ PR_i$. Actually, we present the proof as a derivation of this equation from the defining equation (1). The calculation is concise, machine checkable, and human readable at the same time.

We use ‘$\cdot$’ for function composition. Function application is denoted as usual by a space ‘ ’, having the highest priority, but also by ‘.’ having the lowest priority (which is written as ‘$\$’ in Haskell).

Here is a calculation in which each step tries to manipulate the expression towards the form of an application of mapReduce. Along the way we invent a suitable reducer and mapper. (Of course, this calculation is heavily inspired by our informal exposition above.)

$$PR_{i+1}$$

= viewing a function as a list of (input, output)-pairs

$$[(m, PR_{i+1} m) \mid m \leftarrow nodes]$$

= defining equation (1)

$$[(m, \ demp (\ sum \left[PR_{i+1} n \mid n \leftarrow nodes; m \in out n\right]) \mid m \leftarrow nodes]$$

= define reducer($m, xs$) = ($m, \ demp (\ sum xs)$)

$$[reducer(m, \left[PR_{i+1} n \mid n \leftarrow nodes; m \in out n\right]) \mid m \leftarrow nodes]$$

= map law

$$reducer \ast \ [\left[reducer(m, \left[PR_{i+1} n \mid n \leftarrow nodes; m \in out n\right]) \mid m \leftarrow nodes]$$

= specification groupByKey

$$reducer \ast \ groupByKey \ast [\left[reducer(m, \left[PR_{i+1} n \mid n \leftarrow nodes; m \in out n\right]) \mid m \leftarrow nodes]$$

= definition concat

$$reducer \ast \ groupByKey \ast \ concat \ast [\left[reducer(m, \left[PR_{i+1} n \mid n \leftarrow nodes; m \in out n\right]) \mid m \leftarrow nodes]$$

= list law and out $n \subseteq nodes$

$$reducer \ast \ groupByKey \ast \ concat \ast [\left[reducer(m, \left[PR_{i+1} n \mid m \leftarrow out n\right]) \mid n \leftarrow nodes]$$

= define mapper($n, p$) = ($m, \left[PR_{i+1} n \mid m \leftarrow out n\right]$)

$$reducer \ast \ groupByKey \ast \ concat \ast [mapper(n, PR_i n) \mid n \leftarrow nodes]$$

= map law

$$reducer \ast \ groupByKey \ast \ concat \ast \ mapper \ast \ [\left[n, PR_i n\right] \mid n \leftarrow nodes]$$

= definition mapReduce

$$mapReduce \ mapper \ reducer \ast [\left[n, PR_i n\right] \mid n \leftarrow nodes]$$

= viewing a function as a list of (input, output)-pairs

$$mapReduce \ mapper \ reducer \ast PR_i$$