Implementing the Reordered PageRank Algorithm in Giraph

Verslag ten behoeve van het
Delft Institute of Applied Mathematics
als onderdeel ter verkrijging

van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE
TECHNISCHE INFORMATICA

door

Jorik Oostenbrink

Delft, Nederland
Juli 2015

Copyright © 2015 door Jorik Oostenbrink. Alle rechten voorbehouden.
BSc verslag TECHNISCHE WISKUNDE/INFORMATICA

“Implementing the Reordered PageRank Algorithm in Giraph”

Jorik Oostenbrink

Technische Universiteit Delft

Begeleiders

Dr. ir. M.B. van Gijzen
Dr. M.M. de Weerdt

Overige commissieleden

Dr. ir. F.J. Vermolen
Dr. ir. F.F.J. Hermans

Juli, 2015

Delft
Abstract

PageRank, a method to rank web pages objectively and mechanically, models a random web surfer. The PageRank values $\pi$ can be found by solving $\pi^T G = \pi^T$ and $||\pi^T||_1 = 1$ with $G$ being the transition matrix related to the random web surfer. This problem has a unique solution.

The PageRank problem is most easily solved iteratively, using the power method, where $\pi^{(k)} = \pi^{(k-1)} G$.

In this paper the reordered PageRank algorithms are discussed. These algorithms (proposed by A. N. Langville and C. D. Meyer in [1]) see the PageRank problem as a linear system of equations and begin by reordering the input Graph/matrix. This way only a smaller problem has to be solved. A disadvantage is that it does take a few extra steps to gain $\pi$ from the solution to this smaller problem.

We’ve developed a suitable stopping condition for these algorithms. However, numerical experiments indicate that this stopping condition is much stricter than the stopping condition for the power method.

The reordered PageRank algorithms and the power method have been implemented in Giraph, an open source version of Pregel. Pregel and Giraph are frameworks for solving large graph problems distributively in a vertex centered manner. Because of some of the bugs and features in Giraph (and the inherent complexity of the reordered PageRank algorithms), implementing the reordered PageRank algorithms is much more complicated than implementing the power method.

The reordered PageRank algorithms are not faster than the power method in Giraph. Even when accounting for the difference in stopping condition the power method is much faster than the reordered PageRank algorithms.

1 Introduction

As a result of a query, search engines like Google return a list of web pages sorted on their relevance and importance. Google uses PageRank to rank pages on their importance, a method invented in the nineties by Google founders Sergey Brin and Larry Page to rank web pages objectively and mechanically [2]. PageRank sees the web as a directed graph, where every page is a node and every link an edge. So it can be used to rank the nodes of a directed graph in any domain. These days it is not only used by Google to rank pages, but also in Twitter’s user recommendation service [3], bibliometrics, information network analysis, etc [4].

PageRank models a random web surfer. The surfer starts on a page and then keeps clicking on random links until it (with probability $1 - \alpha$) decides to jump to another page based on a distribution $v$. This continues indefinitely. Jumping does not only make sense intuitively (most people don’t keep clicking links when surfing, but also type in URLs), it also ensures that the PageRank scores exist and are unique.

The random surfer is essentially a random walker, with the PageRank scores $\pi$ (a probability vector, meaning that all elements of $\pi$ are nonnegative and $||\pi||_1 = 1$) being the stationary distribution of the related Markov chain. This in turn can be seen as an eigenvalue problem with eigenvalue 1. The PageRank problem is most easily solved iteratively, using the power method, where $\pi^{(k)} = \pi^{(k-1)} G$. [5]

As the graphs on which PageRank is used can be very large, computing the scores can be very time consuming. This should be dealt with, because the web constantly changes and PageRank values need to be calculated quickly enough to still be useful.

In “A Reordering for the PageRank Problem” A. N. Langville and C. D. Meyer suggest a different way to look at the problem [1]. They first rewrite the problem as a linear system and then reorder the graph matrix, resulting in algorithms which in the most time consuming steps only use a (still quite large) submatrix of the graph matrix.
A good way to speed up the computation of the PageRank values is to calculate them on many machines in parallel. This can be done using a general purpose framework, but it may be preferable to use a framework specifically build to solve large graph problems distributively. In 2010 Google revealed such a framework called Pregel [6]. In their model, programs are expressed as a sequence of iterations on vertices. In each iteration, a vertex can receive messages from the previous iteration, send messages to other vertices for the next iteration, modify its own and its outgoing edges state and mutate graph topology. This model allows for easy scaling and parallelization, and their framework can process very large graphs extremely quickly.

Unfortunately, Pregel isn’t freely available outside of Google. Luckily there does exist an open-source counterpart of Pregel called Giraph [7]. Giraph works in roughly the same way as Pregel, so many ideas should transfer between Giraph and Pregel.

To the best of our knowledge, there is no known approach using the efficiency gains from [1] in calculating PageRank in parallel. In this report we’ll answer the following question:

**Are the reordered PageRank algorithms faster than the power method in Giraph?**

As stated before, computation speed is very important in calculating PageRank. If the reordered PageRank algorithms aren’t faster than the power method then it’s better to use the power method, but if the reordered PageRank algorithms are faster then it might be worth it to consider using them instead.

We’ll compare the computation speeds by implementing both the power method and the reordered PageRank algorithms in Giraph and using them to calculate the PageRank values for two web crawls. This will be done on a single HP EliteBook 8560w, as due to time constraints it wasn’t possible to compare the implementations on multiple machines. However, if the parallel performance of both implementations is comparable, then the results on a single machine are representative of the results on multiple machines.

To this end we’ll first need to answer the following subquestions:

**What is a good stopping condition for the reordered PageRank algorithms?** This question is important for any implementation of the reordered PageRank algorithms, even those not using Giraph. Because the algorithms don’t calculate \( \pi^{(k)} \) every iteration, but a slightly different value, computing the same stopping condition as used in the power method every iteration would take far too much time. We need a stopping condition that is at least as strict as the stopping condition for the power method, but doesn’t take too long to calculate.

**How do you properly implement the reordered PageRank algorithms in Giraph?** Pregel and Giraph are used by Google and Facebook with much success [6,8], so it is interesting to see how one would implement the reordered PageRank algorithms in these frameworks. A good implementation does not only have to give the correct answer, it should also calculate the answer in the least amount of time possible and have good parallel performance (i.e., has to scale well to multiple machines). Both of these last requirements are hard to assess, so we’ll require that the implementations runtime is at most a multiple of the theoretical minimum runtime of the reordered PageRank algorithms (which means that the implementation has to run in linear time in respect to the number of non-zeros in the input matrix) and that its theoretical parallel performance is comparable to that of the power method.

In Section 2 the PageRank problem and the power method will be described in more detail while in Section 3 the reordered PageRank algorithms are discussed and the stopping condition
is determined. The implementation will be detailed in Section 4 and evaluated in Section 5. In Section 5 we’ll also evaluate the stopping condition and compare the speed of the reordered PageRank algorithms to that of the power method. The conclusions will be drawn in Section 6.

2 PageRank and the Power Method

This section describes the PageRank problem and provides a proof for the existence and uniqueness of its solution. At the end, an iterative way to find a numerical solution called the power method will be specified.

The construction of the PageRank problem starts with the $n \times n$ hyperlink matrix $H$. Where $n$ is the number of pages. Every row and column of $H$ represent a page and every element a link between them. If page $i$ links to page $j$, then $H_{ij} = \frac{1}{l_i}$, where $l_i$ is the total amount of outgoing links (outlinks) of page $i$. All other entries are 0. For example, the graph from figure 1 results in the matrix $H = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 0
\end{pmatrix}$.

Figure 1: A directed graph. Taken from [5]

$H$ is not necessarily a valid transition matrix yet, as it can contain rows with all zeros. These rows represent the pages with no outlinks, called the dangling nodes. The random surfer can’t exit these nodes via links, so they have to be dealt with in some way. This can be done in multiple ways, resulting in different PageRank scores. Brin and Page remove the dangling nodes and add them again at the end of the process. But in this paper each dangling node row is replaced with the probability vector $v^T$ (the personalization vector, it will be elaborated on further later). Whatever solution is used, the matrix without zero rows is called $S$. In our case,

$$S = H + dv^T$$

(1)

Where $d$ is a column vector that identifies the dangling nodes. Continuing with the previous example, if $v$ is the uniform probability vector, then

$$S = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 0
\end{pmatrix} + \begin{pmatrix}
0 \\
0 \\
\frac{1}{4} \\
1
\end{pmatrix} \begin{pmatrix}
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4}
\end{pmatrix} = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4}
\end{pmatrix}$$

$S$ is a valid transition matrix, but it does not yet model the jump between pages. If we would use $S$ to calculate the PageRanks, there would be so called rank sinks: Clusters of pages only linking to each other, and thus hoarding all the PageRank. $S$ does not even necessarily have a unique stationary distribution. The random surfer jumps with chance $1 - \alpha$ to a page based on
the probability vector $v > 0$, meaning that every element of $v$ is $> 0$. This can be modelled by adding $(1 - \alpha)v^T$ to every row of $\alpha S$.

The factor $\alpha$ is called the damping factor, as it dampens the effect of $v$. If $\alpha = 0$ then the web-graph part has no effect on the PageRank, the web-graph part becomes more important the closer $\alpha$ gets to 1. It may look tempting to choose an $\alpha$ very close to 1, but not only is it harder to compute the PageRank scores if $\alpha$ is close to 1 (the power method has a convergence rate of $\alpha$ [9]), if $\alpha$ goes to 1, the PageRank scores of the most important nodes go to 0 because of the Rank Sinks. [10] Often, a value between 0.85 and 0.99 is chosen, with 0.85 being the most common.

The vector $v$ is called the personalization vector ($v$ also gets called the teleportation vector, for obvious reasons), because different vectors can be used to personalize the resulting PageRank. For example, if you consider tudelft.nl to be the centre of your online experience, you may want to give it a very high personalization score. This will not only result in tudelft.nl getting a higher PageRank than other pages. Google probably studies, research or the TU Delft to get a higher PageRank than those pages link to too, etc... In this example you might expect pages related to PageRank, pages that tudelft.nl link to will also (indirectly) receive a higher PageRank, and the page tudelft.nl gets to 1. It may look tempting to choose an $\alpha$ between 0 and 1 (for obvious reasons), because different vectors can be used to personalize the resulting PageRank. For example, if you consider tudelft.nl to be the centre of your online experience, you may want to give it a very high personalization score. This will not only result in tudelft.nl getting a higher PageRank than other pages. Google probably studies, research or the TU Delft to get a higher PageRank than other pages. Google probably uses a $v$ based on the TrustRank algorithm [11], designed to weed out spam pages. In research, the uniform probability vector is used most often.

The matrix which models the random surfer completely is called the Google Matrix $G$. Calculated by

$$G = \alpha S + (1 - \alpha)1v^T$$  \hspace{1cm} (2)

$$= \alpha H + (\alpha d + (1 - \alpha))v^T$$  \hspace{1cm} (3)

Where $1$ is a column vector of ones. To finish our example, if $\alpha = 0.85$,

$$G = 0.85 \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix} + 0.15 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{3}{8} & \frac{7}{8} & \frac{3}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{7}{8} & \frac{3}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{7}{8} & \frac{3}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{7}{8} & \frac{3}{8} & \frac{3}{8} \end{pmatrix}$$

Finally, the PageRank vector is defined as the probability vector $\pi$ for which $\pi^T G = \pi^T$. A page $i$ has a PageRank score $\pi_i$.

To prove that $\pi$ exists and is unique, we need Perron’s Theorem. Perron’s theorem states if a matrix $A > 0$, then there is a unique probability vector $p$ with $p^TA = \lambda p^T$, where $\lambda$ is positive and is the largest eigenvalue of $A$ [12]. Because $v > 0$, this is the case for $G$. We also know that because the rows of $G$ sum to 1, $\|p^T G\|_1 = \|p^T\|_1$, so $\lambda$ must be 1. In conclusion, $G$ has a unique probability vector $\pi$ with $\pi^T G = \pi^T$. Thus proving that the PageRank problem has a unique solution.

The power method, which converges to the eigenvector for the dominant (largest) eigenvalue [9], works by starting with a probability vector $\pi^{(0)}$, and then iteratively calculating

$$\pi^{(k)} = \pi^{(k-1)} G$$  \hspace{1cm} (4)

If the starting value $\pi^{(0)}$ can be written as $\pi^{(0)} = \pi^T + c_2 v_2^T + \cdots + c_n v_n^T$ with $v_2, \cdots, v_n$ eigenvectors of $G$ it is easy to prove that the power method converges to $\pi^T$.

**Proof.** As $\lambda_1 = 1$:

$$\pi^{(1)} = \pi^{(0)} G = \pi^T G + c_2 v_2^T G + \cdots + c_n v_n^T G = \pi^T + \lambda_2 c_2 v_2^T + \cdots + \lambda_n c_n v_n^T$$
where $\lambda_2, \cdots, \lambda_1$ are the eigenvalues of $G$. Inductively:
\[
\pi^{(k)} T = \pi^T + \lambda_2^k c_2 v_2^T + \cdots + \lambda_n^k c_n v_n^T.
\]
Because $\lambda_n < \lambda_{n-1} < \cdots < \lambda_1 = 1$
\[
\lim_{k \to \infty} \pi^{(k)} T = \pi^T
\]
So the power method converges to $\pi^T$ if $\pi^{(0)} T$ can be written as $\pi^{(0)} T = \pi^T + c_2 v_2^T + \cdots + c_n v_n^T$.

Using (3), a power iteration can also be expressed as
\[
\pi^{(k)} T = \pi^{(k-1)} T G
= \pi^{(k-1)} T (\alpha (H + dv^T) + (1 - \alpha) 1v^T)
= \alpha \pi^{(k-1)} T H + \alpha \pi^{(k-1)} T dv^T + (1 - \alpha) v^T
\]
So the only vector matrix multiplication necessary is $\pi^{(k-1)} T H$. Because $H$ is a sparse matrix (a matrix containing mostly zeros), this multiplication can be done way quicker than $\pi^{(k-1)} T G$.
Every iteration of the power method costs $O(nnz(H))$ operations, where $nnz(H)$ is the number of non-zeros in $H$.

The convergence rate of the power method is $\frac{|\lambda_2|}{|\lambda_1|}$. Where $\lambda_1$ is the largest and $\lambda_2$ the second largest eigenvalue. The second largest eigenvalue of $G$ is less than or equal than $\alpha$ (Often, if $H$ is based on a web-graph, $\lambda_2 = \alpha$ [9]). So the convergence rate of the power method is $\leq \alpha$. For example, if $\alpha = 0.85$, about $\frac{\log(10^{-8})}{\log(0.85)} \approx 114$ iterations are needed for a convergence tolerance of less than $10^{-8}$.

3 Reordered Pagerank Algorithms

In “A Reordering for the PageRank Problem” [1] A. N. Langville and C. D. Meyer give 2 algorithms to solve the PageRank problem based on viewing the problem as a linear system of equations and then reordering it. We’ll call these algorithms the reordered PageRank algorithms. In this section we will discuss their properties.

Section 3.1 and 3.2 mostly restate information from [1], only changing the proof for the system of linear equations and adding a short explanation for the convergence rate. In Section 3.3 we provide a stopping condition for the reordered PageRank algorithms.

3.1 Theory

Solving the PageRank problem is equivalent to solving the following system of linear equations:
\[
x^T (I - \alpha H) = v^T
\]
where
\[
\pi^T = \frac{x^T}{||x^T||_1}
\]
The following proof is a little different from the proof from [1], but leads directly to the desired answer and also gives us $||x^T||_1$:

$$
\begin{align*}
\pi^T G &= \pi^T \quad \text{(Definition)} \\
\Leftrightarrow \quad 0 &= \pi^T - \pi^T G \\
&= \pi^T I - \pi^T (\alpha H + (\alpha d + (1 - \alpha)1) v^T) \quad \text{(Using (3))} \\
&= \pi^T (I - \alpha H) - (\alpha d + (1 - \alpha)) v^T \\
\Leftrightarrow \quad \pi^T (I - \alpha H) &= (\alpha d + (1 - \alpha)) v^T \\
\Leftrightarrow \quad x^T (I - \alpha H) &= v^T
\end{align*}
$$

Where $x^T = \frac{\pi^T}{\alpha d + (1 - \alpha)}$. From this it immediately follows that $||x^T||_1 = \frac{1}{\alpha d + (1 - \alpha)}$ and $\pi^T = \frac{x^T}{||x^T||_1}$.

It is interesting to note that $x^T = \pi^T$ (its minimum) if and only if every node is a dangling node and that $x^T = \frac{\pi^T}{I - \alpha}$ (its maximum) if and only if there are no dangling nodes.

If we reorder the matrix $H$ so that the rows corresponding to dangling nodes are at the bottom of the matrix we get a matrix with the following form:

$$
H = \begin{pmatrix} H_{11} & H_{12} \\ 0 & 0 \end{pmatrix}
$$

Where $H_{11}$ is a matrix modelling the links between non-dangling nodes and $H_{12}$ the links from non-dangling nodes to dangling nodes.

So the coefficient matrix in the linear system becomes

$$
(I - \alpha H) = \begin{pmatrix} I - \alpha H_{11} & -\alpha H_{12} \\ 0 & I \end{pmatrix}
$$

The inverse of this matrix is

$$
(I - \alpha H)^{-1} = \begin{pmatrix} (I - \alpha H_{11})^{-1} & \alpha(I - \alpha H_{11})^{-1} H_{12} \\ 0 & I \end{pmatrix}
$$

Because $x^T = v^T (I - \alpha H)^{-1}$ we can now write it as

$$
x^T = \left[ v_1^T (I - \alpha H_{11})^{-1} \mid (\alpha v_1^T (I - \alpha H_{11})^{-1} H_{12} + v_2^T) \right]
$$

Where $v^T$ has been divided into a non-dangling ($v_1^T$) and dangling ($v_2^T$) part.

This leads us directly to the first reordered PageRank algorithm.

### 3.1.1 Algorithm 1

**Step 1:** Reorder $H$ so that the rows corresponding to dangling nodes are at the bottom of the matrix.

**Step 2:** Solve $x_1^T (I - \alpha H_{11}) = v_1^T$

**Step 3:** Compute $x_2^T = \alpha x_1^T H_{12} + v_2^T$

**Step 4:** Normalize $\pi^T = \frac{[x_1^T \ x_2^T]}{||x_1^T \ x_2^T||_1}$

This algorithm is equivalent to the iterative method proposed by Lee, Golub and Zenios in [13]. This isn’t strange, as the reordered PageRank algorithms are based on this method.
$H_{11}$ can also contain zero rows, which we can also move to the bottom of the matrix. Resulting in an even smaller upper left submatrix of $H$ on which we can repeat the same process. After recursively reordering smaller and smaller submatrices of $H$ we can write it as:

$$H = \begin{pmatrix} H_{11} & H_{12} & H_{13} & \cdots & H_{1b} \\ 0 & H_{23} & \cdots & H_{2b} \\ 0 & 0 & \cdots & H_{3b} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$ (7)

Where $b \geq 2$ is the number of square diagonal blocks in the reordered matrix.

The coefficient matrix now has the following structure

$$(I - \alpha H) = \begin{pmatrix} I - \alpha H_{11} & -\alpha H_{12} & -\alpha H_{13} & \cdots & -\alpha H_{1b} \\ -\alpha H_{12} & I & -\alpha H_{23} & \cdots & -\alpha H_{2b} \\ -\alpha H_{13} & -\alpha H_{23} & I & \cdots & -\alpha H_{3b} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\alpha H_{1b} & -\alpha H_{2b} & -\alpha H_{3b} & \cdots & I \end{pmatrix}$$

We can now formulate algorithm 2.

### 3.2 Algorithm 2

Step 1: Recursively reorder $H$ till it has the same structure as (7) with $H_{11}$ containing no zero rows.

Step 2: Solve

$$x_1^T (I - \alpha H_{11}) = v_1^T$$ (8)

Step 3: For $i = 2$ to $b$, compute

$$x_i^T = \alpha \sum_{j=1}^{i-1} x_j^T H_{ji} + v_i^T$$ (9)

Step 4: Normalize

$$\pi^T = \frac{[x_1^T x_2^T \cdots x_b^T]}{\|[x_1^T x_2^T \cdots x_b^T]\|_1}$$ (10)

Notice that if we stop early in step 1, then algorithm 1 is equivalent to algorithm 2 with $b = 2$. From now on we will only refer to algorithm 2.

A simple but effective method to solve (8) is with the Jacobi method:

$$x_1^{(k)T} = \alpha x_1^{(k-1)T} H_{11} + v_1^T$$ (11)

Since

$$x_1^T - x_1^{(k)T} = \alpha (x_1^T - x_1^{(k-1)T}) H_{11}$$
This method converges with a convergence rate of
\[
\frac{||x_1^T - x_1^{(k)T}||_1}{||x_1^T - x_1^{(k-1)T}||_1} \leq \alpha ||H_{11}||_1 \leq \alpha
\]
So when dealing with web graphs, this method converges to \(x_1\) with at the least the same rate as the power method converges to \(\pi\).

It is obvious that while step 1 may contain some relatively expensive memory operations, step 2 is the step that takes most of the time. Each iteration of step 2 will cost about \(O(nnz(H_{11}))\) operations. As the number of iterations does not depend on the input, this means that a good implementation of these algorithms has a linear runtime in respect to the number of non-zeros in the input.

### 3.3 Stopping Condition

A good stopping condition for the power method is
\[
||\pi^{(k)T}G - \pi^{(k)T}||_1 < \epsilon
\]

[14], which is (for the power method) equivalent to
\[
||\pi^{(k+1)T} - \pi^{(k)T}||_1 < \epsilon
\]

This conditions is not suitable for the reordered PageRank algorithms as \(\pi^{(k)}\) isn’t readily available at every iteration. To find a good stopping condition for these algorithms we need to find a value \(s\) (and set \(s < \epsilon\)) which can be calculated at almost no cost using only \(x_1^{(k)T}\), such that \(||\pi^{(k)T}G - \pi^{(k)T}||_1 \leq s\). To minimize the amount of iterations, \(s\) should be as close as possible to \(||\pi^{(k)T}G - \pi^{(k)T}||_1\). In this section we will see that
\[
\frac{2}{||x_1^{(k)T}||_1 + 1 - ||v_1^T||_1}||x_1^{(k+1)T} - x_1^{(k)T}||_1
\]
is such a value.

Before we can prove this we first need the following:

**Lemma 1.** For every iteration of the reordered PageRank algorithms:
\[
x^{(k)T} = [x_1^{(k)T} - x_1^{(k+1)T} 0^T] + \alpha x^{(k)T}H + v^T
\]

\[
||x^{(k)T}||_1 = ||[x_1^{(k)T} 0^T] + \alpha (x^{(k)T}H - [x_1^{(k)T}H_{11} 0^T])||_1 + 1 - ||v_1^T||_1
\]

and
\[
||x^{(k)T}||_1 \geq ||x_1^{(k)T}||_1 + 1 - ||v_1^T||_1
\]

**Proof.** Using (9) to rewrite \(x^{(k)T}\) we get:
\[
x^{(k)T} = [x_1^{(k)T} \mid (\alpha \sum_{j=1}^{1} x_j^{(k)T} H_{j2} + v_2^T) \mid \cdots \mid (\alpha \sum_{j=1}^{b-1} x_j^{(k)T} H_{jb} + v_b^T)]
\]

\[
= [x_1^{(k)T} 0^T] + \alpha [x_1^{(k)T} x_2^{(k)T} \cdots x_{b-1}^{(k)T} 0^T] \begin{pmatrix} 0 & H_{12} & \cdots & H_{1b} \\ 0 & H_{23} & \cdots & H_{2b} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & H_{(b-1)b} \\ 0 & \cdots & \cdots & 0 \end{pmatrix} + [0^T v_2^T \cdots v_b^T]
\]

\[
= [x_1^{(k)T} 0^T] + \alpha (x^{(k)T}H - [x_1^{(k)T}H_{11} 0^T]) + [0^T v_2^T \cdots v_b^T]
\]
Combining (17) with (11) results in
\[
x^{(k)T} = [x_1^{(k)T} 0^T] + \alpha x^{(k)T} H - [x_1^{(k+1)T} 0^T] + v^T
\]
\[
= [x_1^{(k)T} - x_1^{(k+1)T} 0^T] + \alpha x^{(k)T} H + v^T
\]

Proving (14).
Knowing that
\[
||[x_1^{(k)T} 0^T] + \alpha (x^{(k)T} H - [x_1^{(k)T} H_{11} 0^T])||_1 \geq 0
\]
and
\[
||0^T v_2^T \cdots v_0^T||_1 = 1
\]
(15) and (16) follow immediately from (17):
\[
||x^{(k)T}||_1 = \|[x_1^{(k)T} 0^T] + \alpha (x^{(k)T} H - [x_1^{(k)T} H_{11} 0^T])||_1 + 1 - ||v_1^T||_1
\]
\[
\geq ||x_1^{(k)T}||_1 + 1 - ||v_1^T||_1
\]

Now we can prove

**Theorem 1.**
\[
||\pi^{(k)T} G - \pi^{(k)T}||_1 \leq \frac{2}{||x_1^{(k)T}||_1 + 1 - ||v_1^T||_1} ||x_1^{(k+1)T} - x_1^{(k)T}||_1
\]

**Proof.** We start by expanding \( G \) in the same way as in (5):
\[
||\pi^{(k)T} G - \pi^{(k)T}||_1 = ||\alpha \pi^{(k)T} H + \alpha \pi^{(k)T} d v^T + (1 - \alpha) v^T - \pi^{(k)T}||_1
\]
We can use (10) to replace \( \pi \) with \( x \):
\[
||\pi^{(k)T} G - \pi^{(k)T}||_1 = \left|\left| \frac{\alpha}{||x^{(k)T}||_1} x^{(k)T} H + \frac{\alpha}{||x^{(k)T}||_1} ||x_b^{(k)T}||_1 v^T + (1 - \alpha) v^T - \frac{1}{||x^{(k)T}||_1} x^{(k)T} \right|\right|_1
\]
Where we also used
\[
x^{(k)T} d = ||x_b^{(k)T}||_1
\]
From (14) it follows that:
\[
\frac{\alpha}{||x^{(k)T}||_1} x^{(k)T} H - \frac{1}{||x^{(k)T}||_1} x^{(k)T} = \frac{1}{||x^{(k)T}||_1} (\alpha x^{(k)T} H - [x_1^{(k)T} - x_1^{(k+1)T} 0^T] - \alpha x^{(k)T} H - v^T)
\]
\[
= \frac{1}{||x^{(k)T}||_1} (x_1^{(k+1)T} - x_1^{(k)T} 0^T - v^T)
\]
Also:
\[
\frac{\alpha}{||x^{(k)T}||_1} ||x_b^{(k)T}||_1 v^T + (1 - \alpha) v^T = \frac{1}{||x^{(k)T}||_1} (\alpha ||x^{(k)T}||_1 - \alpha \sum_{j=1}^{b-1} ||x_j^{(k)T}||_1 + ||x^{(k)T}||_1 - \alpha ||x^{(k)T}||_1) v^T
\]
\[
= \frac{1}{||x^{(k)T}||_1} (||x^{(k)T}||_1 - \alpha \sum_{j=1}^{b-1} ||x_j^{(k)T}||_1) v^T
\]
Finally, using (16) to estimate
\[ \| \pi^{(k)} T G - \pi^{(k)} T \|_1 \] = \frac{1}{\|x^{(k)} T\|_1} \| [x_1^{(k+1)} T - x_1^{(k)} T \mathbf{0}^T] + (\|x^{(k)} T\|_1 - \alpha \sum_{j=1}^{b-1} \|x_j^{(k)} T\|_1 - 1) v^T \|_1 \]
\leq \frac{1}{\|x^{(k)} T\|_1} \left( \| [x_1^{(k+1)} T - x_1^{(k)} T \mathbf{0}^T] \|_1 + \|x^{(k)} T\|_1 - \alpha \sum_{j=1}^{b-1} \|x_j^{(k)} T\|_1 - 1 \| v^T \|_1 \right) \]

It follows from [15], [11],
\[ \sum_{j=1}^{b-1} \|x_j^{(k)} T\|_1 = \|x^{(k)} T H\|_1 \]
and
\[ \| v^T \|_1 = 1 \]
that:
\[ \| [x^{(k)} T]_1 - \alpha \sum_{j=1}^{b-1} \|x_j^{(k)} T\|_1 - 1 \| v^T \|_1 = \| [x^{(k)} T]_1 - \alpha (x^{(k)} T H - [x_1^{(k)} T H_1] \mathbf{0}^T)] \|_1 = \| v^T \|_1 - \alpha \|x^{(k)} T H\|_1 \]
\leq \| [x^{(k)} T]_1 + \alpha (x^{(k)} T H - [x_1^{(k)} T H_1] \mathbf{0}^T)] - v^T - \alpha x^{(k)} T H\|_1
= \| [x^{(k)} T - x^{(k+1)} T \mathbf{0}^T] \|_1 \]

So
\[ \| \pi^{(k)} T G - \pi^{(k)} T \|_1 \leq \frac{2}{\|x^{(k)} T\|_1} \| [x_1^{(k+1)} T - x_1^{(k)} T \mathbf{0}^T] \|_1 \]
\[ = \frac{2}{\|x^{(k)} T\|_1} \|x_1^{(k+1)} T - x_1^{(k)} T\|_1 \]

Finally, using (16) to estimate \( \frac{1}{\|x^{(k)} T\|_1} \):
\[ \| \pi^{(k)} T G - \pi^{(k)} T \|_1 \leq \frac{2}{\|x_1^{(k)} T\|_1 + 1 - \| v^T \|_1} \|x_1^{(k+1)} T - x_1^{(k)} T\|_1 \]

So
\[ \frac{2}{\|x_1^{(k)} T\|_1 + 1 - \| v^T \|_1} \|x_1^{(k+1)} T - x_1^{(k)} T\|_1 < \epsilon \] (18)

is a valid stopping condition for the reordered PageRank algorithms.

Remark (A Priori Stopping Condition). Of course, it’s also possible to use the convergence rates to determine when to stop before even starting to calculate the PageRank. For example, we already showed that for the power method (if \( \alpha = 0.85 \)) about \( \frac{\log(10^{-8}) \alpha}{\log(0.85)} \approx 114 \) iterations result in a convergence tolerance of less than \( 10^{-8} \).

In the same way, it’s also possible to find the number of iterations resulting in certain convergence tolerances for the reordered PageRank algorithms. (Although you would need to find a way to connect \( \| \pi^T - \pi^{(k)} T \|_1 \) to \( \|x_1^T - x_1^{(k)} T\|_1 \).)

This had the advantage that every iteration takes less time to compute, as the stopping condition does not have to be recalculated every iteration. This does not only impact the computation.
time directly, it also means that less messages have to be send between machines, improving parallel performance.

However, most of the time the required precision has been reached before the estimated number of iterations. Meaning that using a predetermined stopping condition results in calculating more iterations than necessary. For this reason we choose to not use a predetermined stopping condition.

4 A Parallel Implementation in Giraph

After finding the stopping condition we can now implement the reordered PageRank algorithms in Giraph.

The first part of this section explains what Giraph is and how you can implement custom programs to be executed by Giraph called Giraph jobs. After that the implementation of the Power method and the reordered PageRank algorithms will be treated in that order. At the end a few difficulties in implementing the Reordered PageRank algorithms in Giraph will be noted.

4.1 Giraph

Giraph [7] is a distributed iterative graph processing framework built on top of Hadoop [15] (using ZooKeeper [16] to elect a master node). Each vertex is assigned to a worker and the workers can be spread among different nodes/machines.

In every iteration, called a superstep, computations are done on every active vertex. These can include sending messages to other vertices, changing its own or its outgoing edges value and mutating the graph by adding or removing vertices and edges. Every action only takes effect in the next superstep, so for example, vertices only receive messages sent in the previous superstep. This goes on until every vertex has voted to halt (thus marking itself as inactive) or computations are halted centrally.

Before every superstep, a central master computation gets executed on the master node. In this computation aggregated and reduced values from the previous superstep are available and messages can be broadcast to all vertices. It is also possible to decide to halt all further computations.

You can start a Giraph job by assigning which classes to use for certain tasks, like handling input or doing computations. These classes should implement specific interfaces or extend specific base classes. This allows you to combine classes to form your own job makes Giraph very modular, although it is mostly useful for using different input and output types.

To implement a custom job, Giraph requires you to extend certain abstract classes and/or implement certain interfaces offering function prescriptions. For every job at least the Computation interface should be implemented. This is normally done by extending AbstractComputation and implementing the compute method, which takes a vertex and messages as input and gets executed once per active vertex every superstep. It is important to note that Computation objects get recreated every superstep, so they can’t be used to store data between supersteps. The only way data can be stored between supersteps is in the vertex values (or in MasterCompute).

For executing master computations (or for additional setup) the MasterCompute class should be extended and the compute and initialize methods implemented. Compute takes no input and gets executed before every superstep. Initialize gets executed at the start of the job and can be used to register aggregators for example (aggregators are used to aggregate certain values of all vertices). MasterCompute is a Writable, so the readFields and write methods must also be implemented. This allows the MasterCompute to be easily stored and recreated.
Finally, when the available input formats are insufficient, custom input formats can be implemented by extending VertexInputFormat, for reading vertices together with their outedges, or EdgeInputFormat and VertexValueInputFormat, for reading edges and vertices separately. When using custom value types for the vertices and/or edges the Writable interface must be implemented, so they can be easily serialized.

![Visualization of a Giraph Job](image)

Figure 2: Visualization of a Giraph Job

Some other customisations are possible too (like executing a computation on every worker before or after every superstep), but are not used in the implementations described in this paper.

The implementations in this paper were made for and executed with the (at the time of writing) latest official release of Giraph: Version 1.1.0

4.2 Graph Partitioning

Giraph divides vertices across workers, which are divided across machines. To maximize parallel performance, vertices have to be divided in such a way that the least amount of messages get send between different workers. Most of the times, including in this implementation, this is equivalent to minimizing the amount of edges between machines. Because every worker should also get about the same amount of vertices, finding such a partition is quite difficult (even with 2 workers this problem is NP-hard). Much research has been done to approximate such a partition (for example [17, 18]), but the problem is outside the scope of this paper.

We use the default partitioning of Giraph, which assigns each worker vertices based on vertex IDs, completely ignoring edges. If both implementations have an otherwise comparable parallel performance, this should not affect the comparison between their runtimes.

4.3 Power Method

The implementation of the power method in Giraph consists of 2 classes (excluding Writables and input formats): PowerMethodComputation and PowerMethodMasterCompute.

PowerMethodComputation extends BasicComputation, which is a subclass of AbstractComputation that receives the same message type as it sends. PowerMethodComputation only accepts vertices with a LongWritable as id, a PairWritable as value and edges without any values. PairWritable is a custom Writable that contains 2 doubles: the PageRank and the Personalization value of a vertex. PowerMethodComputation sends and receives DoubleWritables as messages.
PowerMethodMasterCompute extends DefaultMasterCompute, which is a subclass of MasterCompute that does nothing. This has the advantage that readFields and write do not have to be implemented, as PowerMethodMasterCompute does not store any information.

For the power method we want to compute $\pi$ at every superstep. If every vertex sends its PageRank value to all its neighbors this is equivalent to computing $\alpha \times \text{sum(messages)} + \alpha \times \text{dangling.get()} \times v + (1 - \alpha) \times v$ for every vertex. Alpha is the damping factor, which can be set by the user when starting the job, sum(messages) is the sum of all incoming messages, dangling.get() is the aggregated sum of the pageRank values of the dangling nodes and $v$ is the personalization value of the vertex.

After computing their PageRank values every non-dangling vertex sends its PageRank to all its neighbours while dangling vertices send their PageRank to the DANGLINGSUM aggregator.

The sum of the absolute differences in PageRank value between steps gets aggregated by the DIFSUM aggregator. Before every superstep PowerMethodMasterCompute checks if the sum of these differences (e.g. $||\pi^{(k)} - \pi^{(k-1)}||_1$) is less than epsilon (which can be set by -ca epsilon=value) and halts all computations if true.

PowerMethodComputation:

```java
compute(vertex, messages) {
    double pageRank;
    if (getSuperstep() >= 1) {
        PairWritable value = vertex.getValue();
        double v = value.getV();
        double previousPageRank = value.getPageRank();
        DoubleWritable dangling = getAggregatedValue(DANGLINGSUM);
        pageRank = alpha * sum(messages) + alpha * dangling.get() * v + (1 - alpha) * v;
        vertex.setValue(new PairWritable(pageRank, v));
        aggregate(DIFSUM, new DoubleWritable(abs(pageRank - previousPageRank)));
    } else {
        pageRank = vertex.getValue().getPageRank();
    }

    // Dangling nodes add their PageRank to DANGLINGSUM
    if (vertex.getNumEdges() == 0) {
        aggregate(DANGLINGSUM, new DoubleWritable(pageRank));
    } // All other nodes send their PageRank to all adjacent vertices
    else {
        sendMessageToAllEdges(vertex, new DoubleWritable(pageRank / vertex.getNumEdges()));
    }
}

PowerMethod:

compute() {
    if (getSuperstep() >= 2) {
        DoubleWritable difSum = getAggregatedValue(DIFSUM);
        if (difSum.get() < epsilon) {
            haltComputation();
        }
    }
}
4.4 Reordered PageRank Algorithm(s)

The reordered PageRank algorithms calculate $\pi$ in 4 steps. So the implementation does too. Each step uses 1 or more computation classes which get assigned by the MasterCompute class ReorderedMasterCompute. ReorderedMasterCompute keeps track of the current step with a variable called step and the current superstep within each step with a variable called subStep.

It is also possible to pre- and post-process the input and output so that only step 2 is executed in Giraph or to execute a different Giraph job for every step.

Pre- and post-processing the input and output (i.e. perform step 1, 3 and 4 outside Giraph) so that only step 2 is executed in Giraph makes the program almost equivalent to the implementation for the power method and much less complex. Because Giraph isn’t well suited to this kind of removal and recreating of vertices (see Difficulties) it could also result in a speed boost. A disadvantage is that to not gain extra runtime in comparison to executing all 4 steps in Giraph, step 1, 3 and 4 still have to be executed in a distributed way. Which means that another framework has to be found and used to do this. Using 2 different parallel frameworks aside each other may cause some difficulties. The biggest reason that this implementation does not only execute step 2 in Giraph is that the whole point was to implement the reordered PageRank algorithms in Giraph. Only executing step 2 would not only result in a very simple Giraph implementation, it would also completely go against this goal.

This implementation does a lot of unnecessary work on vertices that could just be set to inactive. Unfortunately, vertices deactivated by voteToHalt() (the only way to temporarily deactivate vertices) can only be reactivated by receiving a message and using voteToHalt() may even lead to the halting of the whole job (if all vertices are halted). This means that it isn’t suitable to use for a program that executes multiple steps in one job. As we’d want vertices to wake up when a new computation class is set and we don’t want the job to halt until step 4 is finished.

It is highly suitable for when all steps are executed in their own job however. Unfortunately, this is still not worth it. As setting up a new Giraph job (reading the input, getting nodes from Hadoop, etc) and shutting down a Giraph job (writing output, releasing Hadoop nodes, etc) both take a large amount of time. So doing this 4 times just to calculate the PageRank just isn’t worth it, even with the speed gain from deactivating vertices.

All vertices use a LongWritable as id, a QuadWritable as value and have edges without any values. A QuadWritable holds 2 doubles and an int: $x$ (or the PageRank), the personalization value, and the starting number of edges of the vertex. It can also hold removed vertices that the vertex should recreate later on. It is necessary to store the initial number of edges a vertex has because $x$ has to be divided by the initial and not the current number of edges when sending it in step 2 and 3. The type of messages sent differs per computation class.

The first step uses the ReorderComputation class to iteratively remove dangling nodes and form $H_{11}$. Every superstep all vertices send their ids to their neighbours. This way the dangling nodes know their incoming edges. All dangling nodes remove themselves from the graph and create a RemovedVertex object containing all the necessary data to recreate them and send it to the vertex that should. They have to remove themselves by first marking themselves to be removed (by setting its numEdges value to -1) and then removing themselves in the next superstep. As Giraph automatically creates a new vertex if you send messages to a non-existing vertex. Or in this case, a removed vertex. The dangling nodes also have to notify all vertices with edges pointing towards them that they should remove those edges. As the dangling nodes can’t remove them themselves. Because if multiple vertices remove outedges from the same vertex in the same superstep this results in an error. If a dangling node has no incoming edges, vertex -1 gets assigned to recreate the dangling node in step 3. (It is assumed that the input
does not contain negative IDs. If a vertex marks itself to be removed it aggregates a true to
the REORDERING aggregator, so that ReorderedMasterCompute knows that the graph is still
being reordered.

As ReorderComputation needs to send LongWritables and RemovedVertices. A special
Writable LongOrRemovedVertexWritable has been created which can contain either a Long-
Writable or a RemovedVertexWritable.

The reordering stops (and step 2 begins) when no dangling nodes have been removed in the
previous superstep or when the number of removal supersteps have been larger than maxReorder
(which can be set by -ca maxReorder=value). Notice that because in the first superstep only id’s
get exchanged and no vertices are removed, this is not when subStep is equal to maxReorder, but
one step later. The minimum value of maxReorder is 1 in which case the implementation executes
algorithm 1. The default value is Integer.MAX_VALUE, essentially resulting in algorithm 2.

ReorderComputation:

```java
compute(vertex, messages) {
  if (vertex.getValue().getNumEdges() == -1) {
    // Vertex has set its value to -1 to indicate it should be removed
    removeVertexRequest(vertex.getId());
    return;
  }
  if (getSuperstep() >= 2) {
    processMessages(vertex, messages, getSuperstep());
  }
  if (vertex.getId() == -1) {
    // Vertex -1 exists only to recreate vertices with no incoming edges
    // And should not remove itself
    return;
  }
  LongOrRemovedVertexWritable idMessage = new LongOrRemovedVertexWritable(vertex.getId());
  sendMessageToAllEdges(vertex, idMessage);
  if (getSuperstep() == 0) {
    // Set the correct number of edges
    QuadWritable value = vertex.getValue();
    vertex.setValue(new QuadWritable(value.getX(), value.getV(), vertex.getNumEdges()));
    // Add the vertex -1 to the graph
    addVertexRequest(new LongWritable(-1), new QuadWritable(0, 0, 1));
  } else if (vertex.getNumEdges() == 0) {
    // When the vertex has received IDs (incoming edges) and is a dangling node: remove it
    ArrayList edges = new ArrayList();
    LongWritable negID = new LongWritable(-vertex.getId().get());
    for (LongOrRemovedVertexWritable message : messages) {
      if (message.isLongWritable() && message.getLong().get() > 0) {
        edges.add(message.getLong().get());
        // Send the negative id to signal that this edge should be removed
        sendMessage(message.getLong(), new LongOrRemovedVertexWritable(negID));
      }
    }
    if (edges.isEmpty()) {
      // When the vertex has no incoming edges, assign vertex -1 as owner
      // And create an edge between them
      edges.add(-1);
    }
  }
  QuadWritable value = vertex.getValue();
  RemovedVertex me = new RemovedVertex(vertex.getId().get(), value, edges);
```
// Assign edges as owner
sendMessage(new LongWritable(edges.get(0)), new LongOrRemovedVertexWritable(me));
aggregate(REORDERING, new BooleanWritable(true));

// We can't remove the vertex immediately,
// because it will still receive messages next superstep
vertex.setValue(new QuadWritable(-1, -1, -1));
}
}
processMessages(vertex, messages, superstep) {
ArrayList removedVertices = new ArrayList();
for (LongOrRemovedVertexWritable message : messages) {
if (message.isLongWritable()) {
    if (message.getLong().get() < 0) {
        vertex.removeEdges(new LongWritable(-message.getLong().get()));
    }
} else {
    removedVertices.add(message.getVertex());
}
}
if (!removedVertices.isEmpty()) {
    vertex.setValue(vertex.getValue().put(superstep - 2, removedVertices));
}
}
ReorderedMasterCompute:

step1Compute() {
(BooleanWritable notDone = getAggregatedValue(REORDERING);
if (subStep >= 2 && (!notDone.get() || subStep - 1 >= reorderSteps.get(getConf()))) {
    // Number of reorder steps depend on if it was stopped by maxReorder or not
    if (!notDone.get()) {
        numReorderSteps = subStep - 2;
    } else {
        numReorderSteps = subStep - 3;
    }
    setComputation(SwitchFrom1Computation);
    step = 2;
    subStep = 0;
}
}

The most work in step 2 gets done by the XComputation class, which computes $x_1$. But XComputation sends DoubleWritables as messages and ReorderComputation sends LongOrRemovedVertexWritable. So in the first superstep of step 2 SwitchFrom1Computation gets used to switch between message types. Because not all the marked vertices may have been removed and not all messages processed yet, SwitchFrom1Computation also removes the marked vertices from the graph and processes the incoming messages. And as $||v_{1T}^T||_1$ is necessary to check for the stopping condition, SwitchFrom1Computation aggregates this value as well (using the SIZE aggregator).

SwitchFrom1Computation:

calculate(vertex, messages) {
if (vertex.getValue().getNumEdges() == -1) {
    // Vertex has set its value to -1 to indicate it should be removed
    removeVertexRequest(vertex.getId());
    return;
}
ReorderComputation.processMessages(vertex, messages, getSuperstep());

QuadWritable value = vertex.getValue();
if (vertex.getNumEdges() != 0) {
    sendMessageToAllEdges(vertex, new DoubleWritable(value.getX() / value.getNumEdges()));
}
aggregate(SIZE, new DoubleWritable(value.getV()));

In every superstep of step 2 all vertices calculate their x value by $\alpha \times \text{sum}(\text{messages}) + v$ and send it to all neighbours.

The aggregator DIFSUM gets used to sum all absolute differences between the previous value of $x_1$ and the current one $(||x_1^{(k)} - x_1^{(k-1)}||_1)$ and the aggregator SUM gets used to sum all values of $x_1$. So that ReorderedMasterCompute can switch to step 3 when (13) is less than epsilon (which can be set by -ca epsilon=value).

XComputation:

\[
\text{compute}(\text{vertex}, \text{messages}) \{ \\
\quad \text{double } x; \\
\quad \text{QuadWritable value = vertex.getValue();} \\
\quad \text{double } v = value.getV(); \\
\quad \text{double } \text{previousX = value.getX();} \\
\quad x = \alpha \times \text{sum}(\text{messages}) + v; \\
\quad \text{vertex.setValue(value.setX(x));} \\
\quad \text{aggregate(DIFSUM, new DoubleWritable(Math.abs(x - previousX)));} \\
\quad \text{aggregate(SIZE, new DoubleWritable(previousX));} \\
\quad \text{if (vertex.getNumEdges() != 0)} \{ \\
\quad \quad \text{sendMessageToAllEdges(vertex, new DoubleWritable(x / value.getNumEdges()));} \\
\quad \}
\}
\]

ReorderedMasterCompute:

\[
\text{step2Compute()} \{ \\
\text{if (subStep == 1)} \{ \\
\quad \text{DoubleWritable sum = getAggregatedValue(SIZE);} \\
\quad \text{vSum = sum.get();} \\
\quad \text{setComputation(XComputation);} \\
\} \text{ else } \{ \\
\quad \text{DoubleWritable difSum = getAggregatedValue(DIFSUM);} \\
\quad \text{DoubleWritable size = getAggregatedValue(SIZE);} \\
\quad \text{double } \text{stopValue = 2 / (size.get() + 1 - vSum) * difSum.get();} \\
\quad \text{if (stopValue < epsilon)} \{ \\
\quad \quad \text{setComputation(RestoreComputation);} \\
\quad \quad \text{broadcast(RESTORE, new LongWritable(numReorderSteps));} \\
\quad \quad \text{step = 3;} \\
\quad \text{subStep = 0;} \\
\} \\
\}
\]

In step 3 all vertices get recreated while at the same time calculating $x$ as a whole. This is done by RestoreComputation, which lets every vertex recalculate their $x$ value if it received
messages. Then RestoreComputation makes every vertex send its \( x \) value to all its neighbours and remove all its edges (so only the values of new vertices get calculated in further steps). After this the owners of the removed vertices recreate them and their incoming edges. Assigning every RemovedVertex to its own vertex spreads the load of recreating them somewhat. Every vertex knows from which superstep to recreate the vertices because ReorderedMasterCompute broadcasts this value every superstep. Notice that because in the first superstep messages from XComputation are received and no edges have been removed yet \( x_1 \) gets recalculated in the first two steps. This results in an end result that is more accurate than strictly necessary. Step 4 begins 3 supersteps after the last vertices have been recreated, when all of \( x \) has been calculated.

**RestoreComputation:**

```java
compute(vertex, messages) {
    LongWritable in = getBroadcast(RESTORE);
    QuadWritable value = vertex.getValue();
    double x = value.getX();
    if (messages.iterator().hasNext()) {
        double alpha = ALPHA.get(getConf());
        double v = value.getV();
        x = alpha * sum(messages) + v;
        vertex.setValue(value.setX(x));
    }
    if (vertex.getNumEdges() != 0) {
        sendMessageToAllEdges(vertex, new DoubleWritable(x / value.getNumEdges()));
    }
    if (in != null) {
        long index = in.get();
        RemovedVertexListWritable removedVertices = vertex.getValue().getRemovedVertices().get(index);
        if (removedVertices != null) {
            for (RemovedVertex v : removedVertices.get()) {
                addVertexRequest(new LongWritable(v.getId()), v.getValue());
            }
        } else if (vertex.getId().get() == -1) {
            // All vertices have been recreated, so vertex -1 should remove itself
            removeVertexRequest(vertex.getId());
        }
    }
}
```

**ReorderedMasterCompute:**

```java
step3Compute() {
    if (numReorderSteps - subStep >= 0) {
        broadcast(RESTORE, new LongWritable(numReorderSteps - subStep));
    } else if (numReorderSteps - subStep == -3) {
        // 3 steps later, to allow the final \( x_b \) to compute it's value
        setComputation(FinalComputation);
        step = 4;
        subStep = 0;
    }
}
```
Step 4 only takes 2 supersteps. In the first one FinalComputation sums all the values of $x$ using the SIZE aggregator, resulting in $||x||_1$. In the final superstep every vertex divides its $x$ value by $||x||_1$ (resulting in $\pi$) and votes to halt. ReorderedMasterCompute does not have to do anything in step 4.

FinalComputation:

```java
compute(vertex, messages) {
    DoubleWritable size = getAggregatedValue(SIZE);
    QuadWritable value = vertex.getValue();
    if (size.get() == 0) {
        aggregate(SIZE, new DoubleWritable(value.getX()));
    } else {
        vertex.setValue(value.setX(value.getX() / size.get()));
        vertex.voteToHalt();
    }
}
```

4.5 Automated Testing

This project does not contain any automated test code. Everything is manually tested by running the implementations on a small constructed and a large real graph, comparing the results with the known answer.

These implementations are very suitable for this way of testing. Because if one step contains a bug the final answer will be incorrect. The specific step containing the bug can then be easily found by outputting the results after each step. The only step for which this is not necessarily applicable is step 1, as not removing all dangling nodes does not result in a wrong answer. That’s why as an additional test the results of step 1 get looked at for the small problem, for which it is known how many nodes should be removed in which step.

This method has the advantage of making a large difference in implementation time, as the scale of the project is quite small in relation to the necessary complexity of the code that would be required to test it. It does have the disadvantage of requiring more work to change it in the future, as after every change the code has to be manually retested. However, it is not expected that this project will be changed (or even used) much in the future, as it is purely intended to show how one can implement the reordered PageRank algorithms in Giraph and to compare runtime.

4.6 Difficulties

There were numerous difficulties in implementing these algorithms in Giraph.

For one, there is a severe lack of documentation, and the documentation that is available is often outdated. In this case using it may even result in the wrong outcome.

There are also numerous bugs in Giraph. For example, Giraph can’t handle hostnames containing upper-case letters. But the biggest bug that had to be worked around is the occasional failure of removeEdgesRequest in large graphs. This did not only take a long time to find, but also requires that all edges be removed directly by the vertices themselves.

A feature that caused much trouble is that if a message gets send to a non-existing vertex, this vertex gets created instead of the message being ignored. Working around this feature does not only make the implementation way more complex, it also often requires extra supersteps to be executed just to send messages (to not conflict with new vertices, as sending messages to a newly created vertex results in an error) or remove vertices one superstep later.
5 Evaluation and Numerical Experiments

Having implemented the algorithms in Giraph, we can now compare the runtime of the power method with that of the reordered PageRank algorithms. But first we will evaluate the theoretical performance of the implementations and compare the convergence rate of the Reordered Algorithm with to the convergence rate of the power method and the stopping condition (18) with (12). At the end of this section we will compare the run time of the reordered algorithms and the power method in Giraph.

5.1 Theoretical Parallel Performance

For good parallel performance it is important that the workload is spread evenly among devices and there aren’t many messages send between devices. So we can give some insight into the theoretical parallel performance just by stating how good the workload is spread across vertices and how many messages are send between them.

5.1.1 Power Method

In the power method the workload is spread very evenly among vertices, with a slight difference in workload (by calculating sum(messages)) depending on the number of incoming edges. Vertices send messages to all outgoing edges. The power method also uses 2 aggregators. Aggregators result in a lot of extra messages, as they do not only require some messages to be send between workers to aggregate values, but the resulting aggregated value is also send to every vertex (regardless of whether they use it or not).

5.1.2 Reordered PageRank Algorithm

**Step 1** In step 1 vertices not only send messages along their outgoing edges. Dangling nodes also send messages along their incoming edges, with the RemovedVertex messages being quite large. This is compensated a little for by the fact that step 1 only uses 1 aggregator. On top of that this aggregator aggregates booleans, so the resulting messages are very small. The workload depends on if a vertex is a dangling node and on the number of incoming messages.

**Step 2** The workload in step 2 is spread very evenly among the different vertices, with (just as in the power method) a slight difference in workload depending on the number of incoming edges. Step 2 of the reordered PageRank algorithm is also very close to the power method in terms of messages being sent, as vertices send messages to all their neighbours and two aggregators are being used.

**Step 3** The workload of step 3 is spread relatively evenly among vertices if and only if the responsibility for recreating vertices is spread evenly among the vertices. But even then, the workload isn’t spread perfectly, as newly created vertices have to do more work (calculating their $x$ value) than other vertices. The number of messages being sent among vertices is pretty low, because vertices only send messages to their newly created neighbours. But ReorderedMasterCompute does send a broadcast message to all vertices.

**Step 4** In step 4 no messages are being send between vertices and the workload of all vertices is exactly the same. The only messages being send are caused by aggregating $\|x^T\|_1$. Step 4 has the best parallel performance of all steps.
We are ignoring the messages being sent to vertices by aggregators from other steps however. In every step every vertex receives the aggregated value from all 3 aggregators, even when they aren’t being used. (This is another argument for splitting the steps among different jobs, although it probably is still not worth it)

The parallel performance of the reordered PageRank algorithms implementation seems to be comparable to that of the power method. But this can’t be said for sure without testing on multiple machines.

5.2 Theoretical Runtime

For both the power method and the reordered PageRank algorithms, the number of iterations does not depend on the size of the input. So we can find their theoretical runtime (on one machine) by checking the theoretical runtime of their supersteps.

5.2.1 Power Method

For a superstep of the power method, summing all incoming messages and sending messages between vertices take the most time. Both summing all incoming messages and sending messages to all neighbors for all vertices takes \( O(\text{number of edges}) \) time. But the number of edges in the graph is equal to the number of non-zeros in \( H \), so the theoretical runtime of the power method is \( O(nnz(H)) \).

5.2.2 Reordered PageRank Algorithms

**Step 1** In step 1, processing (and creating) messages take the most time. Messages in step 1 are only send across edges, so the runtime of step 1 is \( O(nnz(H)) \).

**Step 2** In step 2, summing all incoming messages and sending messages between vertices take the most time. So the runtime of step 2 is \( O(nnz(H_{11})) \).

**Step 3** In step 3, recreating edges, sending messages across edges and summing these messages take the most time. So the runtime of step 3 is \( O(nnz(H)) \).

**Step 4** In step 4 we only sum \( x \) and then divide \( x \) by \( ||x||_1 \). If \( H \) is a \( n \times n \) matrix. This only takes \( O(n) \) time.

All in all, the runtime of the reordered PageRank algorithms implementation is \( O(nnz(H)) \), which means that the implementation passed our requirements as stated in the Introduction.

5.3 Test Cases

For the numerical experiments 2 matrices downloaded from the Gleich group in the UF Sparse Matrix Collection [19] were used as Graph: The 9,914 \( \times \) 9,914 \( \text{wb-cs-stanford} \) matrix and the 820,878 \( \times \) 820,878 \( \text{flickr} \) matrix. The personalization vectors used are the uniform probability vectors and \( \alpha = 0.85 \).

5.4 Convergence rates and the Stopping Condition

The convergence rates have been compared in Matlab by computing \( ||\pi^{(k)T}G - \pi^{(k)T}||_1 \) for the first 20 iterations of the reordered PageRank algorithms and the power method on the Stanford
matrix. These values were plotted against each other and against the stopping condition we found earlier.

Figure 3: The first 20 iterations of the reordered PageRank algorithm 1 and the power method.

As seen in figures 3 and 5, the power method converges faster than the reordered PageRank algorithms. This may seem in contrast with the convergence rate found for the reordered PageRank algorithms ($\leq \alpha$, while in this case the power method should have a convergence rate of $\alpha$), but this was the convergence rate for $x_1$ and not for $\pi$.

There is also quite a big difference between $||\pi^{(k)T}G - \pi^{(k)T}||_1$ and the stopping condition value [13]. This was expected as [13] is a rough estimation of $||\pi^{(k)T}G - \pi^{(k)T}||_1$ using only $x_1^{(k)T}$. But it does mean that the implementation of the reordered PageRank algorithm stops later than necessary.

As seen in figure 1, the difference between $||\pi^{(k)T}G - \pi^{(k)T}||_1$ for both reordered PageRank algorithms is very small, probably because the difference in size of $H_{11}$ is also quite small. $H_{11}$ is a $7053 \times 7053$ matrix in algorithm 1 and a $6585 \times 6585$ matrix in algorithm 2. As expected, $||\pi^{(k)T}G - \pi^{(k)T}||_1$ is smaller for algorithm 2 than for algorithm 1, because the convergence rate to $x_1$ and so also the convergence rate to $\pi$ depends on $H_{11}$.
Figure 4: The first 20 iterations of the reordered PageRank algorithm 2 and the power method

Figure 5: The difference between $||\pi^{(k)}^T G - \pi^{(k)}^T||_1$ for both reordered PageRank algorithms
5.5 Measured Runtime

The implementations were run on a HP EliteBook 8560w running Hadoop 3.5.2 and Zookeeper 3.4.6 using only one worker, so this is a comparison of non-parallelized runtime. This means that there is a lot of extra overhead (because Giraph is meant to work with multiple workers on different machines/nodes) for no gain. However, it does allow us to see the minimum total amount of processing time necessary across all nodes, with the runtime on multiple nodes being runtime on 1 node

SwitchFrom1Computation counts as a part of step 2 in this comparison. Epsilon was set to its default value (0.0001).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Reordered PageRank 1</th>
<th>Reordered PageRank 2</th>
<th>Power Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1 (Supersteps)</td>
<td>2</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 1 (Runtime)</td>
<td>1,140 ms</td>
<td>2,912 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 2 / computing π</td>
<td>47</td>
<td>47</td>
<td>33</td>
</tr>
<tr>
<td>Step 2 / computing π</td>
<td>16,173 ms</td>
<td>13,969 ms</td>
<td>10,088 ms</td>
</tr>
<tr>
<td>Step 3 (Supersteps)</td>
<td>3</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 3 (Runtime)</td>
<td>1,323 ms</td>
<td>5,613 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 4 (Supersteps)</td>
<td>2</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 4 (Runtime)</td>
<td>4,124 ms</td>
<td>591 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Total (Supersteps)</td>
<td>54</td>
<td>65</td>
<td>33</td>
</tr>
<tr>
<td>Total (Runtime)</td>
<td>22,760 ms</td>
<td>23,085 ms</td>
<td>10,088 ms</td>
</tr>
</tbody>
</table>

Figure 6: wb-cs-stanford \((9,914 \times 9,914)\)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Reordered PageRank 1</th>
<th>Reordered PageRank 2</th>
<th>Power Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1 (Supersteps)</td>
<td>2</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 1 (Runtime)</td>
<td>12,366 ms</td>
<td>31,263 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 2 / computing π</td>
<td>45</td>
<td>45</td>
<td>34</td>
</tr>
<tr>
<td>Step 2 / computing π</td>
<td>191,403 ms</td>
<td>184,880 ms</td>
<td>151,960 ms</td>
</tr>
<tr>
<td>Step 3 (Supersteps)</td>
<td>3</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 3 (Runtime)</td>
<td>17,723 ms</td>
<td>17,997 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 4 (Supersteps)</td>
<td>2</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>Step 4 (Runtime)</td>
<td>1,550 ms</td>
<td>1,615 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Total (Supersteps)</td>
<td>52</td>
<td>59</td>
<td>34</td>
</tr>
<tr>
<td>Total (Runtime)</td>
<td>223,042 ms</td>
<td>235,755 ms</td>
<td>151,960 ms</td>
</tr>
</tbody>
</table>

Figure 7: flickr \((820,878 \times 820,878)\)

While operating on the Stanford matrix almost every superstep (regardless of the algorithm used) took around 300 ms, although some supersteps in step 1 and step 3 cost around 600 ms. There were a few strange exceptions which took around 3-4 second. These exceptions didn’t happen on every run (although they happened most of the time), but when they happened they seemed to occur on the same superstep every time. This means that they probably weren’t caused by the laptop performing other tasks, but by something in Java, Hadoop or Giraph, like the garbage collector.

A runtime of around 300 ms per iteration seems to be a kind of minimum, as even with a graph of size 4 no iteration gets much lower. This means that on small graphs it is completely useless to use the reordered PageRank algorithms in Giraph, as the runtime only depends on the number of supersteps. However, Giraph itself is only useful for very large graphs anyway, so this doesn’t matter that much.
In the results of the flickr graph you can see that iterating on a smaller matrix in step 2 does make a difference. Every iteration on the power method takes an average of about 4469 ms. Every iteration of step 2 of algorithm 1 takes an average of about 4253 ms. And every iteration of step 2 of algorithm 2 takes an average of about 4108 ms. This difference is not enough to make the reordered PageRank algorithms worthwhile however, as the amount of extra supersteps required still results in a larger runtime. The power method, which only takes 34 supersteps, is faster than algorithm 1, which takes 52 supersteps and is in turn closely followed by algorithm 2, which takes 59 supersteps.

You may think that the difference between the stopping conditions (18) and (12) is the cause, but even if step 2 of algorithm 1 and 2 only took 34 supersteps their total time would still be $223,042 - \frac{11}{35} 191403 \approx 176,255$ ms and $235,755 - \frac{11}{35} 184,880 \approx 190,562$ ms respectively. So even with a (more than) perfect stopping condition the power method would still outperform the reordered PageRank algorithms in Giraph.

You may also think that using a smaller epsilon value might result in the reordered PageRank algorithms winning from the power method, as an iteration of step 2 of the reordered PageRank algorithms takes less time than an iteration of the power method. However, because of the higher number of iterations needed, even step 2 of both reordered PageRank algorithms takes more time than the power method. Looking at figures 3 and 5 the difference in iterations needed only grows larger when epsilon gets smaller, so even at smaller epsilon values the power method would be faster than the reordered PageRank algorithm.

There may be a sweet spot where if epsilon is in a certain range and if using a stricter stopping condition that does not take too long to calculate the reordered PageRank algorithms are faster than the power method, but looking at the large difference between $||\pi^{(k)T}G - \pi^{(k)T}||_1$ in figures 3 and 5 this is improbable.

We showed before that the parallel performance of the reordered PageRank algorithms shouldn’t be much better than that of the power method, so the power method should also outperform the reordered PageRank algorithms when computing PageRank in parallel.

6 Conclusion

We found a stopping condition for the reordered PageRank problem using only $x_1^{(k)T}$:

$$\frac{2}{||x_1^{(k)T}||_1 + 1 - ||v_1^{(k)}||_1} ||x_1^{(k+1)T} - x_1^{(k)T}||_1 < \epsilon$$

However, this stopping condition is quite a lot stricter than the stopping condition of the power method.

We also properly implemented both the power method and the reordered PageRank algorithms in Giraph.

After doing numerical experiments, it seems that the reordered PageRank algorithms aren’t faster than the power method in Giraph. In contrast, the power method is much faster.

For future work, changing Giraph itself to be more suitable for the reordered PageRank algorithm may be worthwhile. This could be done by, among other things, turning off the automatic creation of non-existent vertices which receive messages and allowing vertices to be deactivated until a superstep has been reached or the computation class is switched.

It would also be interesting to better study the parallel performance of the algorithms by running them in parallel on multiple devices in stead of just one laptop.
References


[16] https://zookeeper.apache.org
